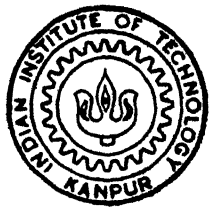
# Mode Transitions

by

**Sanjeev Narayan Khadilkar**






DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

**INDIAN INSTITUTE OF TECHNOLOGY, KANPUR**

October, 1993

# Mode Transitions

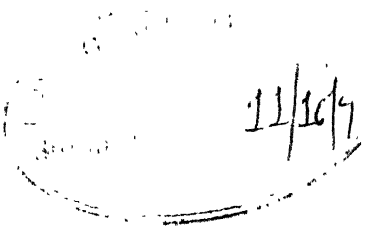
A Thesis Submitted
In Partial Fulfilment of the Requirements
for the Degree of
Doctor of Philosophy

by
Sanjeev Narayan Khadilkar

to the
Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR
October 1993


CSE-1993-D-KHA-MOD

# CERTIFICATE



It is certified that the work contained in the thesis titled *Mode Transitions*, by *Sanjeev Narayan Khadilkar* has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Somenath Biswas

Somenath Biswas
Professor,
Department of Computer Science and Engineering,
Indian Institute of Technology, Kanpur

October 1993

# SYNOPSIS

**Name of student** : Sanjeev Narayan Khadilkar **Roll No** : 8721161
**Degree for which submitted** : Doctor of Philosophy
**Department** : Department of Computer Science and Engineering
**Thesis title** : Mode Transitions
**Name of Thesis Supervisor** : Professor Somenath Biswas
**Month and Year of Submission** : October 1993

## Statement of Results

We propose a modification in the definition of reversals, a complexity measure on the deterministic Turing machine. To distinguish our definition from the classical definition [Hartmanis1968] [KamedaVollmar1970], we refer to our definition as identifying a new resource which is a systolically strict variant of reversals and which we call mode transitions. With mode transitions as the analogue of parallel time and workspace as the analogue of hardware, we give an "efficient" simulation of the DTM on the PRAM. In fact, while the parallel time used is polynomially related to mode transitions (not an improvement over [Pippenger1979]), we show that simultaneously the hardware used is linearly related to workspace (whereas [Pippenger1979] gives an arbitrary polynomial blowup). At the same time, we show that the simulation of PRAMs on the DTM (which uses mode transitions) is no worse than the previously known simulation (which uses reversals). One implication of this is that sequential-access oriented algorithms that use few mode transitions can be easily translated into "efficient" parallel algorithms. We also explore the implications of the hypothesis that PRAMs can be simulated "efficiently" on the DTM (using mode transitions). We show that this would tighten the correspondence between the DTM and the PRAM so much that it would be comparable to the orthodox interpretation of the Invariance Thesis [VanEmdeBoasHandbook1990] and

we propose a Unified Invariance Thesis to express this fact.

## The approach of this study

For every tape the "mode" of the tape is defined as the pattern of O(1) most recent head movements. (The length of this pattern is the same for all tapes and is called the period or cycle-time of the DTM.) A head movement is said to be expected if it cyclically repeats this pattern. If there are no unexpected head movements, then given the iteration count (of the unidirectional scan) and the modes of all tapes, the locations of all heads are fixed. Every (unexpected) head movement costs separately (it is called a mode transition).

First, the simulation of DTMs on PRAMs is presented. The cases of narrow and shallow computations (in the sense of circuit complexity) are considered separately. For narrow computations, the simulation is straightforward. For shallow computations, a nearly cost-optimal parallel algorithm to compute the transitive closure of bounded-width layered directed acyclic graphs is used.

Next, PRAM computations are expressed in logic. A family of extensions of first order logic is proposed. All extensions are based on a single new operator Y(T,S,V) which takes parameters *time domain* T, *space domain* S and *variance domain* V. Each is one of several domains of different sizes that are part of the structure. For each choice of these parameters, formulae express languages and the extended calculus expresses a complexity class. Each formula has a *time arity*, a *space arity* and a *variance arity*. Certain calculii of this family have a "standard form"; i.e. for every formula of the calculus, one can construct an equivalent formula which has the syntactic properties of the standard form. It is shown that for PRAM computations of interest, an equivalent formula in standard form can be constructed, such that the domain sizes and arities syntactically represent the complexity of the PRAM computation modulo polynomial factors.

Finally, this formula is evaluated on the DTM. The workspace and mode transitions required depend on the domain sizes and arities of the formula. The proof adapts techniques

from research on interconnection networks for parallel computing to Turing machine computations. Note that k-ary predicates over a domain of size n can be viewed as binary strings of length $n^k$ stored on a DTM tape. A k-tuple of variables can be viewed as a position of the tape head. A predefined function like successor can be viewed as a map from head positions to head positions. The key idea is that k-ary predicates can also be viewed as the contents of a one-bit register in a $n^k$-processor fixed connection network. A k-tuple of variables can be viewed as a processor index. A predefined function like successor can be viewed as a data transformation to be achieved by routing messages over the network using the properties of the interconnection functions. The way this key idea is applied here is by interpreting familiar interconnection functions as functions over binary strings and showing that these are efficiently implementable on the DTM.

The result is proved first for *conventional resource bounds*; i.e. resource bounds of the form $O(n^k)$, $O((\log(n))^k)$ and $O^*((\log(n))^k)$ for some integer $k \geq 1$, where n is the size of the input. This establishes the proof technique. Then the terms *constructibility on simultaneous complexity models* and *acceptable resource bounds* are defined and the results are extended to acceptable resource bounds.

## Application to database query processing

The paradigms of parallel algorithm design are applied to recursive query processing in databases. Roughly, an efficient recursive query processing algorithm is a polynomial time algorithm which performs only O(polylog(|R|)) file operations on a database consisting of a relation R and stored as a file of records (with a suitable definition of "file operation": an index or sort command on a file of size n takes O(log(n)) file operations while a sequence of next-record commands or a sequence of previous-record commands or a rewind command takes one file operation.) Efficient algorithms are not possible for arbitrary queries. However, since a directed graph with vertices restricted to outdegree 1 may be thought of as a functional dependency, and very efficient parallel algorithms for finding least common ancestors

in such graphs are known, this is possible at least in some special cases. For one subclass of queries, an efficient sequential algorithm that uses few file operations is obtained by analogy with the corresponding parallel algorithm.

# ACKNOWLEDGEMENTS

I would like to thank Professor Somenath Biswas, my thesis supervisor, for his thoughts, advice and for his never ending patience. Without his able guidance this thesis would never have got through. His sincerity, dedication to his work, his enthusiasm and absolute forthrightness in his dealings with people are qualities I greatly admire. I am indebted to him for the constant support—logistic, moral and intellectual—that I received from him throughout, notwithstanding periods when I think he was vexed with me and often justifiably so. I am grateful to him for having borne my idiosyncrasies during the seven years of graduate school.

I would also like to thank Prof. Yuri Gurevich, Dr. Harish Karnick, Dr. Ratan Ghosh, Dr. T. V. Prabhakar, Dr. Pratul Dublish, Vijayan Rajan and Kalyan Basu for formal and informal instruction and for many discussions we had. My interaction with them helped me acquire a feel for diverse subjects that later melted into an unexpected synthesis. I am grateful to Prof. Eric Allender, Prof. David Barrington and Prof. P. S. Thiagrajan for comments on an earlier draft of my thesis.

I am grateful to all my teachers, starting with my parents, for sharing their wisdom. I would like to thank my colleagues and friends at IIT Kanpur for their company, help, inspiration and support. In particular, I am grateful to R. Sundarrajan (Zero Zero Shivan), Jayateerth Mangsuli, Alok Sharan and Kalyan Basu for their help with logistic matters. I also thank the department for the facilities and the infra-structure provided by them. Thanks are also due to the technical staff and the software engineers of the department who put in their efforts to keep the departmental facilities running.

My parents have been very encouraging and supportive while my Ph.D. research was going on. I am grateful to them for the inculcation of right attitude which is my rudder in the voyage of life. This work I dedicate in loving memory to Ajoba (**Prof. C. H. Khadilkar**) and Bapumama (**Mr. C. V. Bhide**), the two Chintamanis who always urged me to become a "professor". To my dear wife Neha, so enthusiastic about and supportive of my attempts at post-doctoral research, belongs my future.

# Contents

# Chapter 1

# Introduction

## 1.1 History and Motivation

In complexity theory, we fix a model of computation and specify the complexity measures to be used. Though equivalence through mutual simulation enabled the treatment of the notion of (un)solvability as a model-independent mathematical principle (Church's Thesis), models which were shown to be computationally equivalent turned out to have differing properties with respect to specific complexity measures (typically, space and time). In the last three decades, model-independent complexity notions have come to light which depend on equivalence through mutual simulation within resource-bounded overhead. Analogous to Church's Thesis for computability theory, we have the Invariance Thesis for complexity theory [VanEmdeBoasHandbook1990]: "Reasonable" machines can simulate each other within a polynomially bounded overhead in time and simultaneously (as per the orthodox interpretation) a constant-factor overhead in space. A variant of Turing's original model, deterministic Turing machines with a two-way read-only input tape, multiple two-way read-write worktapes and a one-way write-only output tape (abbreviated to "Turing machine" hereafter), turned out to be equivalent in the sense of the Invariance Thesis to an idealised von Neumann computer, the random access stored program machine with logarithmic cost (abbreviated to "RAM" hereafter)[VanEmdeBoasHandbook1990] and this has encouraged the rapid adap-

tation of ideas from computability theory (efficient universal machines, recursion schemas, program transformations) to computing systems [ValiantHandbook1990].

The situation in parallel computing is quite different. Practical parallel computing has evolved from research on telephone networks, fast Fourier transforms, sorting, celluler logic and fault-tolerant multiprocessing ([Stone1971], [Feng1974], [Lawrie1975], [Davio1981], [WuRosenfeld1981] and the references therein), and has only recently crystallised into a coherent study of general-purpose parallel architectures [Siegel'sBook1990] [ValiantHandbook1990] [Blelloch'sBook1990]. A variant of the parallel random access machine (PRAM) [ValiantHandbook1990] and a parallel vector scan machine (PVSM) [Blelloch'sBook1990] have been proposed as "appropriate aspiration(s) for the parallel computer architect much as the von Neumann model is in the sequential case" [ValiantHandbook1990]. Resource-bounded simulations between parallel computation models have been studied by many researchers [Siegel'sBook1990] [KarpRamachandranHandbook1990] [ValiantHandbook1990] [Blelloch'sBook1990]. In one approach, the number of processors available P is fixed. If a program written to run in t(n) parallel time with p(n) processors runs in T parallel time when simulated on a P processor machine, the efficiency of the simulation is said to be a lower bound on the ratio (p(n)t(n))/PT. The aim is to achieve constant efficiency for any P $\leq$ p(n)/log(n) [ValiantHandbook1990]. In another approach, the following definition of an "efficient" parallel algorithm is used [KarpRamachandranHandbook1990]: If the fastest sequential algorithm for a problem runs in time O(T(n)) and a given parallel algorithm A runs in parallel time t(n) with p(n) processors, then A is efficient if t(n)=O(polylog(n)) and the work w(n)=p(n)t(n) is O(T(n)polylog(n)). Simulations between some parallel computation models make the notion of an efficient algorithm invariant with respect to the particular model used. A third approach shows that certain parallel computation models are approximately equivalent in that if a given problem can be solved on one of these models in parallel time T(n) using P(n) processors, then it can be solved on any of the other models in parallel time poly(T(n)log(P(n))) using poly(P(n)T(n)) processors. Some pairs of models are equivalent in a tighter sense: their ability to solve problems in $O(\log(n)^k)$ parallel time using polynomially

bounded hardware is identical for each fixed positive integer k [KarpRamachandranHandbook1990]. Some problems are known to be in NC, the class of problems which can be solved in polylog parallel time using polynomially bounded hardware [Pippenger1979] [Cook1985], but are not known to have NC algorithms that are within a polylog factor of the best known lower bound on the work (=processor-time product) [CoppersmithWinograd1982]. Thus, there are three commonly studied degrees of parallelism [Pippenger1987]: the case of a single or a fixed number p of processors, called the serial case, the case of a processor-time product (nearly) equal to the time bound of the best known sequential algorithm or the best known lower bound on such an algorithm, called the balanced case and the case of a number of processors large enough (while still polynomially bounded or at least reasonable in the sense of the Parallel Computation Thesis; see below) to allow the computation to be performed in the minimum possible parallel time, called the highly parallel case.

Much of the work in resource-bounded simulations between parallel computation models can be classified into these three categories. The fine structure of the class NC is usually studied in terms of the classes $NC^k$, $k \geq 1$, where $NC^k$ is the class of problems solvable in $O(\log(n)^k)$ time using polynomially bounded hardware. $NC^k$ as defined above is not robust, and there has been some discussion about the "correct" model on which $NC^k$ (particularly for $k=1$) is to be defined ([Borodin1977], [Ruzzo1981], [Cook1985], [CookHoover1985], [Allender1985], [Allender1989], [ComptonLaFlamme1988]). All this work focuses on obtaining minimal achievable parallel time (i.e. the highly parallel case mentioned above), while permitting any polynomial bound on the hardware. As a result, these efforts, though closer in spirit to classical complexity theory, have remained outside the pale of "efficient" parallel algorithms, and do not appear to have directly contributed to the ongoing research on general purpose parallel architectures and efficient universal parallel machines. (Not one of these papers is cited in [ValiantHandbook1990].) On the other hand, many of the papers which do present "efficient" simulations between parallel computation models tend to draw inspiration as much from computer engineering and algorithm design as from classical complexity theory [Siegel1979], [NassimiSahni1981b], [LevPippengerValiant1981], [Galil-

Paul1983], [BorodinHopcroft1985] [AltHagerupMehlhornPreparata1987],[Blelloch1989]. (All of these are cited in [ValiantHandbook1990].) These, along with some results mentioned in [Valiant1976], [FichRagdeWigderson1988] and [HerleyBilardi1988], have set a defacto minimum standard for theories of practical relevance.

Researchers working on the highly parallel case have made several attempts to compare parallel computation models with sequential ones, particularly the Turing machine [PrattStockmeyer1976] [Borodin1977] [Pippenger1979] [Dymond1980] [Goldschlager1982] [Hong1984a] [Hong1984b] [Parberry1986] [ParberrySchnitger1988] [VanEmdeBoasHandbook1990] [KarpRamachandranHandbook1990]. Two remarkable results ([PrattStockmeyer1976], [Pippenger1979]) have led to the articulation of (respectively) two Parallel Computation Theses, which propose to define "reasonable" parallel computation models as those for which an analogous result can be obtained. Both results show that parallel computation models and Turing machines can simulate each other with polynomially bounded overhead. The simulation of Turing machines on parallel computation models is conceptually similar in the two cases: by representing (partial) ids as nodes and transitions as edges, the problem of simulation is reduced to that of pathfinding in graphs. (The encodings and the simulation overheads differ: [PrattStockmeyer1976] deals with the case when only a space bound is known while [Pippenger1979] deals with the case when a runtime bound as well as additional information in the form of a reversal bound is available.) However the reverse simulations are conceptually different. In [PrattStockmeyer1976] a recursive depthfirst search of the circuit or self-organizing network gives a simulation that uses workspace polynomially bounded by the circuit depth (for stacks) but, because it has to revisit the nodes in general, requires worstcase runtime exponential in the circuit depth even when the circuit size is small. In [Pippenger1979], the nodes of the circuit are represented by fixed tape cells (like callers on a telephone network), while the simulation strategy uses sorting techniques to route packets from source to destination in parallel for all such pairs. This gives a simulation that is iterative rather than recursive: the runtime required is polynomially bounded by the circuit size, the result of simulating one parallel timestep is a continuation that describes the state

of the parallel computation after the step, and the number of reversals is bounded by the product of the number of parallel timesteps and a polynomial in the logarithm of the circuit size. Thus the first approach [PrattStockmeyer1976] enables us to show that parallel time is polynomially related to sequential workspace and the second approach [Pippenger1979] enables us to show that parallel time is polynomially related to reversals and simultaneously parallel hardware is polynomially related to sequential runtime.

However, this work also has not been directly applied to the study of efficient universal parallel machines. (Neither [PrattStockmeyer1976] nor [Pippenger1979] is cited in [ValiantHandbook1990].) The various perspectives have failed to mesh, leading to a widening gap between the "machine-independence" culture of classical theory and the "balanced parallelism" culture of architecture and algorithm design. The hope, that parallelism, interpreted as simultaneous resource bounds, would prove to be a natural continuation of sequential complexity, depended on the anticipation of tight relationships between resources on sequential models and resources on parallel models. Were such a hope to come true, one could argue that sequential and parallel complexity are respectively a one-parameter and a two-parameter analysis of a single phenomenon called computational complexity, and in principle, both can be studied on the same model of computation through inclusion, separation and tradeoff results. The lack of efficient simulations between the DTM and the PRAM in the context of the defacto minimum standard for theories of practical relevance mentioned above, means that either the DTM and the sequential RAM both have to be dropped from the class of "reasonable" models, or, to take the proposals of [VanEmdeBoasHandbook1990] and [ValiantHandbook1990] to their logical conclusion, two separate theories of computational complexity have to be evolved, each with its own resource-bounded version of Church's Thesis, and crudely related to each other through the theorem of [PrattStockmeyer1976], subsequently articulated as the Parallel Computation Thesis.

There have been cases of models acceptable to computability theory proving unreasonable for complexity theory (for example, the unary Turing machine), and the DTM, which has not played an important role in efficient sequential algorithm design, has been less favoured

than the sequential RAM [VanEmdeBoasHandbook1990]. Yet the DTM is the only sequential model that offers even a crude analogue of parallel time (namely reversals); there seems to be no corresponding complexity measure for the sequential RAM. Thus while there have been proposals that the DTM be dropped from the class of "reasonable" models in view of its unsuitability for efficient sequential algorithm design, other sequential models are no better than the DTM when it comes to known efficient mutual simulations with the PRAM. The other alternative (of evolving two independent theories of computational complexity) seems to suffer from what can only be described as a slight overabundance of unprovable computation theses.

## 1.2 Statement of Results

If it turns out that Pippenger's 1979 result can be improved, a unified theory of sequential and parallel complexity might become possible. The key to the result presented here is a change in the definition of reversals. To distinguish our definition from the classical definition [Hartmanis1968] [KamedaVollmar1970], we refer to our definition as identifying a new resource which is a systolically strict variant of reversals and which we call mode transitions. (The formal definition of mode transitions is in chapter 2.) With mode transitions as the analogue of parallel time and workspace as the analogue of hardware, we give an "efficient" mutual simulation of the DTM on the PRAM. In fact, while the parallel time used is polynomially related to mode transitions (not an improvement over [Pippenger1979]), we show that simultaneously the hardware used is linearly related to workspace (whereas [Pippenger1979] gives an arbitrary polynomial blowup). At the same time, we show that the simulation of PRAMs on the DTM (which uses mode transitions) is no worse than the previously known simulation (which uses reversals). One implication of this is that sequential-access oriented algorithms that use few mode transitions can be easily translated into "efficient" parallel algorithms. We also explore the implications of the hypothesis that PRAMs can be simulated "efficiently" on the DTM (using mode transitions). We show that this would tighten the

correspondence between the DTM and the PRAM so much that it would be comparable to the orthodox interpretation of the Invariance Thesis [VanEmdeBoasHandbook1990] and we propose a Unified Invariance Thesis to express this fact.

## 1.3 The Approach of this Study

A step in a machine computation is called a reversal step if one or more heads either shift for the first time or shift in the direction opposite to that in which they last shifted. A head may pause and resume motion in the same direction without there being a reversal step. The resource reversals is defined as the number of reversal steps in a computation.

The key objection to reversals is this: with reversals, there is no restriction on when and how often tape heads pause during an otherwise unidirectional scan. (As has been explicitly specified in [Pippenger1979], the model on which the NC characterisation holds permits the head to pause. It is not clear to us whether a characterisation of NC is at all possible on DTM models in which the head is required to move either left or right on each step.) Because of this, tape heads that start the scan together may not end it together, even when the total length scanned is the same, and the number of distinct head-position configurations that in principle could occur during a unidirectional scan of length s by h tapes is $s^h$ (though at most $h * s$ configurations can occur on any particular scan). Hence the hardware required for the simulation of [Pippenger1979] depends on the $h^{th}$ power of workspace where h is the number of tapes. In [Pippenger1979] the author found it "disturbing that the degree of the polynomial$\cdots$depends on the number of tapes possessed by the simulated machine; (the Hennie and Stearns result [HennieStearns1966] reducing many tapes to two tapes) unfortunately$\cdots$produces an exorbitant increase in reversal."

Our solution is to replace reversals with a new resource which we call mode transitions. The new definition eliminates the key objection to reversals mentioned above. For every tape the "mode" of the tape is defined as the pattern of O(1) most recent head movements. (The length of this pattern is the same for all tapes and is called the period or cycle-time of the

DTM.) A head movement is said to be expected if it cyclically repeats this pattern. If there are no unexpected head movements, then given the iteration count (of the unidirectional scan) and the modes of all tapes, the locations of all heads are fixed. Every (unexpected) head movement costs separately (it is called a mode transition). This eliminates the blowup in the simulation of DTMs on PRAMs because the problem of simulation reduces to computing the transitive closure of layered bounded-width DAGs, which can be solved by an NC algorithm that is within a polylog factor of linear cost.

On the other hand, the definition of mode transitions is flexible enough that the simulation of PRAMs on the DTM is no worse than the previously known simulation (which uses reversals). In particular, the notion of period or cycle-time is designed to allow the heads on different tapes to scan at different "average speeds" by following different left-right rhythms. This makes possible the "efficient" implementation of permutation and broadcast functions used in interconnection networks. A normal form result obtained by a descriptive complexity technique gives a regular structure to the access pattern of PRAM algorithms (at the cost of a polynomial factor blowup in the hardware) and the normalised algorithm is then implemented "efficiently" in terms of the interconnection functions mentioned above.

## 1.4 The Descriptive Complexity Technique

In recent years, much attention has been paid to logics that express various complexity classes. The original approach involved characterisation of (fragments of) conventional logics, e.g. existential second-order logic expresses NP [Fagin1974]. Then attention focused on extensions of first-order logic by new atomic predicates (EQUAL, SUCCESSOR, LESS_THAN, DIVISIBLE_BY_K [folklore], BIT [Fagin1974]), by new kinds of quantifiers and gates (MAJORITY, PARITY and THRESHOLD gates [folklore], COUNTING quantifiers [Immerman1987a] [CaiFurerImmerman1989] [ImmermanLander1990]) by operators that iterate over (syntactically constrained) functionals (e.g. LEAST_FIXED_POINT [AhoUllman1979] [Immerman1982], [Vardi1982], ITERATIVE_FIXED_POINT [GurevichShelah1985], RELA-

TIONAL_PRIMITIVE_RECURSION [ComptonLaFlamme1988]) and by operators that could report the solution to some standard search problem (transitive closure [Immerman1986] or Hamiltonicity [Stewart1989]). Another approach studied expressibility on restricted structures (trees [Lindell1987]). In each case, the effort was to characterise the class of problems that could be described by the logic and to relate the complexity of the problems in this class to the restrictions and the complexity of the operators. This area is referred to descriptive complexity.

An independent research program on the development of new database query languages [Zloof1977], [AhoUllman1979], [ChandraHarel1980], [ChandraHarel1982], [Chandra1988] [AbiteboulVianu1990] [AbiteboulVianu1991a] had initially served as motivation but later descriptive complexity research broadened to include the development of logical senses for complexity notions (inductive depth corresponds to parallel time, number of variables corresponds to number of parallel processors [Immerman1987b]; new atomic predicates in the formalism correspond to relaxing the uniformity condition on circuit families, new types of quantifiers correspond to new types of gates in the circuit [BarringtonImmermanStraubing1988]).

We use the techniques developed by these lines of research to structure the proof of our result simulating PRAMs on DTMs. While it appears to us very difficult to directly simulate the PRAM on the DTM within tight resource constraints, a logic-based "programming language" used as a via media turns out to be both powerful enough to express PRAM computations (though at the cost of a polynomial factor blowup in the hardware) as well as (syntactically) simple enough to permit tightly coded simulations of normalised formulae of this language on the DTM.

The "programming language" we use is first-order logic extended by what we call the Y-operator. Such an extension is usual in descriptive complexity, but here there is one crucial difference in orientation: instead of choosing an extending operator that is of a priori interest, the Y- operator used here has been designed to make the proof simple. The Y-operator constructs a predicate whose arity corresponds to the processor complexity of the

PRAM computation as well as the space complexity of the DTM simulation.

A "standard form" exists for the extended logic FO+Y, consisting of a propositional formula in the scope of a single Y-operator (with some additional nice properties that are of technical importance). Thus we can use the Y-operator indiscriminately while writing a formula expressing the PRAM computation, and then construct the equivalent formula in standard form before simulating it on the DTM.

By constructing a flexible extension of first-order logic, we provide ourselves with a "high-level" programming language in which PRAM computations can be conveniently expressed. By constructing a "standard form", we translate programs in this "high-level" language to a syntactically simplified sub-language that is still equivalent in power. The resulting programs are now simple enough in structure that we can simulate them on the relatively inflexible multitape DTM model in a straightforward manner.

## 1.5 Organization of the Thesis

In Chapter 2, the sequential and parallel models of computation and the resources of interest are defined. Then the resource bounded simulation of the DTM model on the PRAM model is presented. In Chapter 3, the calculus FO+Y(T,S,V) extending first order logic is defined and a standard form is demonstrated. Then the simulation of the PRAM model in the calculus is presented. In Chapter 4, the simulation of the calculus on the DTM model is presented. The main result is presented in Chapter 5. In Chapter 6, the results are summarised, some spin-off possibilities are pointed out and suggestions for future work are given.

# Chapter 2

# Mode Transitions

In this chapter, first the sequential (DTM) and parallel (PRAM) models of computation and the resources of interest are defined. The definition of mode transitions and its motivation appears here. Then the resource bounded simulation of the DTM model on the PRAM model is presented. The simulation of the PRAM model on the DTM model is covered in chapters 3 and 4.

## 2.1 Basic Definitions and Claims

**Definition 2.1.1:**

**The model of sequential computation is the deterministic multi- tape Turing machine having a read-only input tape of finite length (i.e. there are endmarker brackets immediately enclosing the input string beyond which the machine cannot go) and one or more read-write work tapes. W.l.o.g. only decision problems are considered, hence an output tape is unnecessary. Two resources, workspace s(n) and mode transitions x(n) are considered. Workspace is the well-known resource defined as the total number of tape cells accessed by the read/write heads of all worktapes during a computation.**

Mode transitions are defined below.

Definition 2.1.2:

On the multitape DTM model, we define the resource mode transitions as follows:

For each tape, including the input tape, define two terms, viz. the mode of the head and the mode of the tape.

At each instant, the head of each tape is said to be in one of three modes: "LEFT", "STOPPED" and "RIGHT". Before and upto time t=0, all heads are said to be in mode "STOPPED". After each step, each head is said to be in that mode which best describes its motion in that step. (Thus, after the halting state is reached, all heads are said to be in mode "STOPPED".)

Let the alphabet of the machine be $\Sigma$. Define the cycle-time or period of the machine to be $w=\lceil(log_2(|\Sigma|))\rceil$.

At each instant, the mode of each tape is said to be the w-tuple of the modes of the head of that tape at the last w consecutive instants, including the current instant. (Thus, after each step, the new mode of the tape is obtained by a non-cyclic left-shift of this w-tuple, with the new mode of the head as the rightmost element of the new w-tuple. The leftmost (oldest) element of the old w-tuple is lost.)

Two modes of a tape are said to be *similar* if they can be converted into each other by a single cyclic (lossless) shift of the w-tuple. Thus if $m_1,m_2,\cdots,m_{w-1},m_w$ are consecutive modes of the tapehead, the corresponding tape mode $<m_1,m_2,\cdots,m_{w-1},m_w>$ (the "current" mode) is *similar* to $<m_2,\cdots,m_{w-1},m_w,m_1>$ (the expected "next" mode) and to $<m_w,m_1,m_2,\cdots,m_{w-1}>$ (the presumed "previous" mode). When w=1, $<m_1>$ is *similar* only to itself. Note that *similarity* is not an equivalence though the transitive closure of *similarity* is an equivalence. Two modes which are not *similar* are said to be *different.*

A mode transition by a tape at a step is said to occur if the tape is in *different* modes at the instants just before and just after that step. *The mode transitions*

*used in a computation is the total number of steps during that computation at which a mode transition by some tape occurs.*

**Claim 2.1.3:**

*The definition of the resource reversals is obtained from the definition of mode transitions by requiring that (unlike in the case of mode transitions):*

*(1) the mode of the head does not change if the head does not move, and (2) the cycle-time (period) w=1 regardless of the alphabet size,*

*that is, the mode of the head (tape) is "STOPPED" ("<STOPPED>") until the head moves for the first time and is either "LEFT" ("<LEFT>") or "RIGHT" ("<RIGHT>") ever after. Counting the resource as in the case of mode transitions now gives the number of reversals.*

**Definition 2.1.4:**

**A sweep is a contiguous subsequence of the sequence of ids in a computation which begins with either the initial id or the id just after a mode transition on some tape and ends with either the halting id or the id just before a mode transition on some tape (not necessarily the same tape) but has no mode transitions on any tape in between.**

**Claim 2.1.5:**

*The computation sequence of a DTM can be partitioned (uniquely) into a series of sweeps and the number of sweeps is exactly one more than the number of mode transitions used by the computation.*

**Definition 2.1.6:**

**For the duration of a sweep, for each tape head, define the cell number of the cell under the head at the beginning of the sweep to be 0, incrementing to the right for the other cells. Let the mode of the $j^{th}$ tape be denoted mode$_j$ and let the location of the head of tape j after the $i^{th}$ step relative to the location at the beginning of the sweep, which is given by the cell number of the cell under the head, be denoted LOC(i,mode$_j$).**

**Claim 2.1.7:**

*Knowing the mode of the tape during the sweep, the cell number of the cell which would be under the tape head after the $i^{th}$ step of the sweep (provided the sweep lasts that long) can be determined, i.e. LOC() can be computed in time polylog(i+n) by a single processor, where n is the input size.*

**Proof:**

Let mode=$<m_1 \cdots m_w>$. Let the *residual right movement* Del(c), $1 \leq c \leq w$, be defined as (Arr(c)-Ell(c)), where Ell(c) is the number of occurrences of "LEFT" in the first c elements of mode and Arr(c) is the number of occurrences of "RIGHT" in the first c elements of mode. Then LOC(i,mode)=Del(w)$*\lfloor(i/w)\rfloor$+Del(i*mod*w).

∎

**Claim 2.1.8:**

*Knowing the mode of the tape during the sweep and the duration i (number of steps) of the sweep, for each cell number loc, it can be determined in time polylog(i+n) by a single processor, where n is the input size, whether that cell was accessed during the sweep.*

**Proof:**

Let mode=$<m_1 \cdots m_w>$. Let the *maximal movement* Delmax(c) and the *minimal movement* Delmin(c), $1 \leq c \leq w$, be defined as the maximum and minimum over all $1 \leq k \leq c$ of Del(k), where Del(k) is as defined in the proof of the previous claim. Let the duration of the sweep be d. Let the *overshoot* be defined as Over=(Delmax(w)-Del(w)) if $d \geq w$ and Over=0 otherwise. Let the *undershoot* be defined as Under=(Delmin(w)-Del(w)) if $d \geq w$ and Under=0 otherwise. If $Del(w) \geq 0$, the minimum cell number of any cell that is accessed during the sweep is Delmin(min(d,w)) and the maximum cell number of any cell that is accessed during the sweep is Del(w)$*\lfloor(d/w)\rfloor$+max(Over,Delmax(d*mod*w)). If $Del(w) \leq 0$, the minimum cell number of any cell that is accessed during the sweep is Del(w)$*\lfloor(d/w)\rfloor$+min(Under,Delmin(d*mod*w)) and the maximum cell number of any cell that is accessed during the sweep is Delmax(min(d,w)). A cell is accessed during the sweep only if its cell number lies in this range.

∎

**Claim 2.1.9:**

*Knowing the mode of the tape during the sweep and the duration k (number of steps) of the sweep, for each cell numbered (say) j which is accessed during the sweep, the largest i such that LOC(i,mode)=j can be calculated, i.e. the inverse function $LOC^{inv}()$ can be computed in time polylog(k+n) by a single processor, where n is the input size.*

**Proof:**

Let the duration of the sweep be d. We want the largest i not greater than d such that LOC(i,mode)=j. By assumption the cell with cell number j is indeed accessed during the sweep, so such an i exists. W.l.o.g. assume Del(w)≥0 (symmetry) and d≥w (finite modification) and hence also j≥0 (finite modification). We know from previous claims that LOC(i,mode)=Del(w)*⌊(*i*/*w*)⌋+Del(*i*mod*w*) and that the cell number of any cell that is accessed during the first i steps of the sweep is bounded by Del(w)*⌊(*i*/*w*)⌋+Delmax(w). This means that the cell at cell number j is not accessed during the first w*⌊((*j* − *Delmax*(*w*))/*Del*(*w*))⌋ steps of the sweep or after the first w*(1+⌈(*j*/*Del*(*w*))⌉) steps of the sweep. This is a finite range which can be exhaustively enumerated to find the largest i not greater than d such that LOC(i,mode)=j.

∎

**Claim 2.1.10:**

*A DTM that uses at most s(n) workspace and x(n) mode transitions uses at most O((s(n)+n)*x(n)) runtime.*

**Claim 2.1.11:**

*If x(n) is an upper bound on the number of mode transitions used on inputs of length n by a DTM that always halts and this bound is actually achieved for some input for each n, the runtime bound of the DTM is Ω(x(n)). If s(n) is an upper bound on the workspace used by that DTM on inputs of length n, $O(c^{s(n)+\log(n)})$ for some constant c>1 is another bound on the runtime of the DTM. Hence it follows that log(x(n))=O(s(n)+log(n)), for x(n) and s(n) as defined above.*

**Claim 2.1.12:**

*Let x(n) be an upper bound on the number of mode transitions used on inputs of length n by a DTM that always halts. If the amount of workspace used by the DTM on inputs of length n is s(n), s(n)=O(n*c$^{x(n)}$) for some constant c>1, or, equivalently, log(s(n))=O(x(n)+log(n)).*

**Proof:**

Since the DTM always halts, it follows that it works in bounded memory. Again, since it works in bounded memory and halts, no configuration of the DTM other than the halting configuration is ever repeated during a computation. Initially all the worktapes are blank and the input tape is finite and of size n. The only way the DTM can increase its workspace is by moving its worktape head(s) into the blank region beyond what was previously accessed during the computation.

Consider the amount by which the DTM can increase its workspace between two mode transitions, i.e. by one sweep. A mode is a finite pattern of size w, hence the DTM cannot move more than (w/2) steps in the direction opposite to the sweep direction (which is to the right if the residual right movement is positive and to the left otherwise).

The tape cells that have been left behind (are no longer accessible without a mode transition) have no further effect on the computation and the termination of the current sweep is not contingent on either the number or the contents of these tape cells.

If the amount of workspace used by the DTM upto the start of the $i^{th}$ sweep is f(i), and the $i^{th}$ sweep terminates, the maximum number of steps in the $i^{th}$ sweep is O(n+f(i)). During this time, the maximum number of tape cells visited on those worktapes where the tape head is moving into the blank region is O(n+f(i)), hence we have the recurrence f(i+1)=O(n+f(i)) and f(0)=0, where the constant implied by the big-Oh is independent of i and n. Solving this, we get f(i)=O(n*c$^{i}$) for some constant c. With i=x(n), we have f(i)=s(n)=O(n*c$^{x(n)}$).

∎

**Definition 2.1.13:**

**The model of parallel computation is essentially the SIMD PRAM with CRCW(PRIORITY) [Goldschlager1982] (Some details including the placement**

of the input string are different. See below). This is a synchronous parallel machine such that any number of processors may read or write into any word of global memory at any step. If several processors try to write into the same word at the same time, then the lowest numbered processor succeeds. Each processor has a finite set of registers including the following:

PROCESSOR: contains the processor number of the processor. This is a read-only register.

ADDRESS:contains an address of global memory,

CONTENTS:contains a word read from global memory or to be written into it, and

PROGRAM_COUNTER:contains the line number of the instruction to be executed next. The initial content is 1 for all processors and is incremented after each instruction. This register cannot be addressed by any instruction other than BLT and HALT (see below).

The instructions to be simulated are limited to the following:

READ:Read the word of global memory specified by ADDRESS into CONTENTS,

WRITE:Write the word in CONTENTS to the global memory location specified by ADDRESS,

OP $R_a$ $R_b$:Perform operation OP on $R_a$ and $R_b$ leaving the result in $R_b$ (Here OP may be ADD or SUBTRACT. $R_a$ and $R_b$ are register names),

MOVE $R_a$ $R_b$:Move contents of $R_a$ to $R_b$,

INC/DEC R: Increment/Decrement the contents of register R, and

BLT R L:Branch to line numbered L if the contents of R is less than zero. L must be greater than 0.

HALT: Halt all processors whose index is greater than or equal to that of the processor executing the instruction. The content of the program counter register of all halted processors is reset to 0 and cannot change thereafter.

There is a common read-only program memory from which the instructions are read. The computation is said to have halted when all processors have halted.

The input is a binary string of length n. All registers and global memory locations are width(n)=$\lceil(log(n)+2)\rceil$ bits wide. The first global memory location contains the number n. (Integers are stored in sign- magnitude format.) The input string is placed width(n) bits at a time in the next $\lceil(n/width(n))\rceil$ global memory locations. The least significant bit of the second global memory location is the leftmost bit of the input string, and any unused bits in the $(\lceil(n/width(n))\rceil+1)^{th}$ global memory location are 0. All these locations are read-only and attempts to write to them are equivalent to no-ops or skips. All subsequent global memory locations are read/write and their contents are originally 0.

The resources of interest are *hardware* and *parallel time*, denoted respectively as h(n) and t(n), n being the length of the input. Parallel Time is simply the number of synchronous steps in a parallel computation. A parallel computation is said to use p(.) processors if the largest index of any processor activated on inputs of length n is p(n). It is said to use m(.) global memory if the largest address of any global memory word accessed on inputs of length n is $\lceil(n/width(n))\rceil$+1+m(n). (That is, the read-only locations used for supplying the input are free.) The amount of Hardware used by the computation is defined to be h(n)=(p(n)+m(n))*width(n).

When the amount of global memory is not of interest, or when m(n)=O(p(n)), the hardware is linearly bounded by the processor-wordwidth product p(n)*width(n).

Claim 2.1.14:

*The CRAM model of [Immerman1987b] differs from the CRCW(PRIORITY)PRAM model of [Goldschlager1982] only by the addition of special instructions that are of help in sublogarithmic time computations. For the present study, the parallel time is $\Omega(log(n))$, so the two models are equivalent to our model and the results here hold for the CRAM model as*

*well as the CRCW(PRIORITY)PRAM model.*

**Claim 2.1.15:**

*The instantaneous id of a DTM that uses at most s(n) workspace and x(n) mode transitions on any input of length n (the state of the automaton, the contents of all non-input tapes and the head locations and modes of all tapes including the input tape; the input tape contents are excluded since they are part of the read-only input to the PRAM) can be represented using O(s(n)+log(n)) hardware on a PRAM. The amount of hardware is independent of the number of mode transitions x(n).*

**Proof:**

A tape of the DTM is represented by an array in global memory. Each cell of the tape contains w=O(1)-bit data (w is the period or cycle-time of the DTM). Since there are width(n) bits in one global memory word, $\lfloor(width(n)/w)\rfloor$ cells are packed in one location of the array. The location of the head on the tape can be specified in $\lceil(log(s(n)))\rceil$ bits and is represented in $\lceil(log(s(n))/width(n))\rceil$ words of global memory. This is O(1) for s(n)=O(poly(n)). The mode of the tape requires $\lceil(wlog(3))\rceil$ bits and is represented in $\lceil(wlog(3)/width(n))\rceil$=O(1) global memory words. Each tape except the input tape of the DTM is represented in this manner.

For the input tape only the head location and the mode is represented. The state of the machine requires O(1) bits and is maintained in O(1) global memory words. The total amount of hardware used for representing the instantaneous id of the DTM (including the head location and mode of the input tape but excluding the input tape contents) is O(s(n)+log(n)), corresponding to O((s(n)/width(n))+1) global memory locations.

∎

## 2.2 Simulation of the DTM model on the PRAM model

We are interested in simulating simultaneous workspace and mode transition bounded DTMs on simultaneous hardware and parallel time bounded PRAMs. For the pur-

poses of this section, *conventional workspace bounds* s(n) and *conventional mode transition bounds* x(n) are those of the form $O(\log(n)^k)$, $O(n^k)$ and $O^*(n^k)$ (for some integer $k \geq 1$), with the restriction that either $x(n)=\Omega(n)$ and $s(n)=O(poly(x(n)))$ or $s(n)=\Omega(n)$ and $\log(s(n))=O(poly(x(n)+\log(n)))$. ($O^*(f(n))$ is a shorthand for $O(f(n)*polylog(f(n)))$.) This rules out DTMs working in simultaneous o(n) workspace and o(n) mode transitions from consideration here even if they happen to use $\Omega(n)$ runtime. (Extending the following results for this case is open.)

**Theorem 2.2.1:**

*For conventional workspace bound s() and conventional mode transition bound x(), such that $x(n)=\Omega(n)$ and $s(n)=O(poly(x(n)))$, let M be a DTM that uses at most s(n) workspace and x(n) mode transitions on any input of length n and accepts language L. Then there is a PRAM algorithm that uses at most $h(n)=O(s(n)+\log(n))$ hardware and $t(n)=O(poly(x(n)+\log(n)))$ parallel time and accepts L.*

**Proof:**

The input to the simulation consists of the description of the DTM and the input string x in encoded form. Any suitable encoding will do as long as it takes $(w|x|)+O(1)$ bits in all, where w is the period or cycle- time of the DTM. The encoding should be easy to parse since it will be accessed in the innermost loop of the simulation for obtaining the input tape contents and for computing the DELTA function. The instantaneous id of the DTM (including the head location and mode of the input tape but excluding the input tape contents) can be represented using $O(s(n)+\log(n))$ hardware.

The transition function of the DTM is encoded as a subroutine DELTA which assumes that the "current" state and the symbols under the tape heads are available in encoded form in some registers of the processor and returns the "next" state, the symbols to be written and the head movements on the tapes in encoded form in some other registers of the processor. The subroutine DELTA can be executed by any (or all) the processors in parallel possibly with different arguments. The execution of the subroutine itself has no direct effect on global memory. The subroutine may be executed redundantly as required for different phases of the

simulation.

A naive sequential simulation can be carried out on a PRAM using only one processor (i.e. without using any parallel features) in time t(n)=O(poly((s(n)+n)*x(n)+n)) with additional hardware overhead O(log(x(n))+log(s(n))+log(n)). W.l.o.g. we assume log(x(n)=O(s(n)+log(n)) (see basic claims above). Since s(n)=O(poly(x(n))), we have t(n)=O(poly(n*x(n)+n)) and h(n)=O(s(n)+log(n)) including the additional hardware overhead. Also x(n)$\dot{=}\Omega$(n), so we have t(n)=O(poly(x(n))), hence t(n)=O(poly(x(n)+log(n))) as required. Details are omitted.

∎

**Theorem 2.2.2:**

*For conventional workspace bound s() and conventional mode transition bound x(), such that s(n)=$\Omega$(n) and log(s(n))=O(poly(x(n)+log(n))), let M be a DTM that uses at most s(n) workspace and x(n) mode transitions on any input of length n and accepts language L. Then there is a PRAM algorithm that uses at most h(n)=O(s(n)+log(n)) hardware and t(n)=O(poly(x(n)+log(n))) parallel time and accepts L.*

**Proof:**

The input to the simulation consists of the description of the DTM and the input string x in encoded form. Any suitable encoding will do as long as it takes (w|x|)+O(1) bits in all, where w is the period or cycle- time of the DTM. The encoding should be easy to parse since it will be accessed in the innermost loop of the simulation for obtaining the input tape contents and for computing the DELTA function. The instantaneous id of the DTM (including the head location and mode of the input tape but excluding the input tape contents) can be represented using O(s(n)+log(n)) hardware.

The transition function of the DTM is encoded as a subroutine DELTA which assumes that the "current" state and the symbols under the tape heads are available in encoded form in some registers of the processor and returns the "next" state, the symbols to be written and the head movements on the tapes in encoded form in some other registers of the processor. The subroutine DELTA can be executed by any (or all) the processors in parallel possibly

with different arguments. The execution of the subroutine itself has no direct effect on global memory. The subroutine may be executed redundantly as required for different phases of the simulation.

Only t(n)=poly(x(n)+log(n)) parallel iterations are available on the PRAM. Though the DTM time bound in the worst case is O((s(n)+n)*x(n)), the number of sweeps is at most x(n)+1 and the goal is to implement each sweep using a small amount of parallel time. The simulation consists of an initialization phase, a construction phase of x(n)+1 iterations (one iteration per sweep) and a reporting phase. The initialization phase, the reporting phase and each iteration of the construction phase are implemented in poly(x(n)+log(n)) parallel time with O(s(n)+log(n)) hardware, thus satisfying the requirements.

In the initialization phase, all the arrays (except the input) are written with the "blank" symbol, the heads are positioned in the center of the arrays and the modes are set to STOPPED$^w$. (Since it is not known in advance whether the DTM will use the tape to the left of the initial head position or the tape to the right of it, s(n) space is allocated on each side of the initial position. Hence the head is in the center of the array.)

Each iteration of the construction phase consists of a setup stage, a path-finding stage and a commit stage, which together simulate one sweep of the DTM computation. In the setup stage a directed layered (acyclic) graph is constructed which encodes all possible computation sequences during the sweep. In the path-finding stage, using the arrays in global memory, the tape modes, the tape head locations, the DTM-state and the transition function DELTA, the actual computation sequence is constructed as a path in this graph. The use of an efficient parallel algorithm for finding this (long) path quickly leads to the speedup of the sweep. In the commit stage, all the arrays in global memory, the tape modes, the tape head locations and the DTM-state are updated to represent the computation id at the end of the sweep.

Finally, in the reporting phase, the output of the DTM computation (accept/reject) is taken as the output of the PRAM simulation. Essentially, each sweep is treated as a reachability problem on a restricted class of graphs (layered DAGs). In pseudo-code, this algorithm may be summarised as follows:

```
function simulate(tape0:global-array of tape-symbols):bool
/* tape0 is the input tape */
alloc tape1,tape2,···,tapeh:global-array of tape-symbols,
mode0,mode1,mode2,···,modeh:tape-mode,
head0,head1,head2,···,headh:integer, /* location of the head */
state:DTM-state;
begin
mode0←STOPPED^w;head0←0;
for(i=1;i<=h;i++){tapei←BLANK;modei←STOPPED^w;headi←0;}
state←START;
while((state<>ACCEPT)and(state<>REJECT)){
construct-graph; find-path; update-status;
}
if(state==ACCEPT)return(TRUE); else return(FALSE);
end.
```

In the pseudo-code above, the call to procedure construct-graph represents the setup stage, the call to procedure find-path represents the path-finding stage and the call to procedure update-status represents the commit stage. It is now shown how these are implemented.

The vertex set of the graph is constructed as follows: The maximum number of steps in a sweep is equal to the workspace bound w*s(n), which also determines the size of the arrays (n=LIN is the input length). Let Q be the set of states of the machine. Let $\Sigma$ be the tape alphabet. Let $\Sigma^w$ be the set of w-tuples of elements of $\Sigma$. (w is the cycle-time or period of the machine) The most recent w tape symbols read by a tape head (including the symbol currently under the head) can be represented by an element of $\Sigma^w$. Let h be the number of tapes. For the $j^{th}$ tape head, let $a_j \in \Sigma^w$ denote the most recent w symbols read by the $j^{th}$ tape head, as above. For each $i{:}0 \leq i \leq s(n)$, each $b \in Q$ and each $j{:}1 \leq j \leq h$, introduce a vertex with label $<i,b,a_1,\cdots,a_j,\cdots,a_h>$ in the graph.

The interpretation is as follows: Each vertex with label (say) $<i,b,a_1,\cdots,a_j,\cdots,a_h>$ repre-

sents a hypothesis that just after the $i^{th}$ step of the sweep, the machine is in state b, relative to the beginning of the sweep the location of the $j^{th}$ tape head is LOC(i,mode$_j$) and the most recent w symbols read/scanned by the $j^{th}$ tape head is $a_j$, the symbol currently being scanned being last($a_j$) (Note that the symbols *written* during the last w steps are not explicitly represented in the label). The computation id just after the $i^{th}$ step of the sweep is correctly represented by an initially unknown such vertex. Identifying, for each i, precisely which vertex is the representative one is the goal. Let this representative vertex be denoted LAB(i). (The vertices are named by their labels.)

Under the hypothesis that the representative vertex just after the $i^{th}$ step of the sweep is u (LAB(i)=u), the label of the representative vertex v just after the $(i+1)^{th}$ step of the sweep (LAB(i+1)=v) can be determined from the transition function DELTA and the label of u, together with the initial conditions of the sweep (i.e. the machine state, tape contents and head positions at the beginning of the sweep and the tape modes during the sweep). In the graph introduce an edge from u to v to represent this relationship, which we denote as a function NEXT(u)=v. Since the machine is deterministic, the outdegree of each vertex is at most 1.

NEXT(.) can be implemented efficiently as a subroutine. To see this, suppose $u=<i,b,a_1,\cdots,a_j,\cdots,a_h>$. Knowing the initial conditions of the sweep, we can compute for each tape j the last w locations of the head of tape j as LOC(i-w+1,mode$_j$),$\cdots$,LOC(i,mode$_j$). The symbols scanned at the last w steps are given by the elements of w-tuple $a_j \in \Sigma^w$, $1 \leq j \leq h$ in the label of u. Using the current state b and the symbols currently being scanned, the subroutine DELTA returns the symbols to be written and the head movement if any on each tape. The head movements specified by DELTA enable us to compute the new locations of all the heads.

We claim that these new locations are either being accessed for the first time during the sweep or are one of the last w locations computed above. (The cells left well behind cannot be accessed again during the same sweep.) If the location is being accessed for the first time, we obtain the new symbol under the head by looking up the global array. If it is one of

the last w locations, we determine the last time that location was accessed, recompute the symbol that was *written* at that step, and take that as the new symbol under the head. Thus we can construct the label of the next vertex and this is v.

As a bonus, we compare the head movements specified by DELTA with the known modes of the tapes, and this tells us whether a mode transition has occurred during this step. If it has, u is the last vertex of this sweep and v=NEXT(u) is the first vertex of the next sweep.

The correct representative vertices LAB(i), $0 \leq i \leq (w*s(n))$, form a directed path that represents the computation sequence. The vertex representing the start of the sweep can be easily identified and is designated the start vertex s. The number of vertices is O(s(n)), and the graph is acyclic and layered (edges go only from one layer to the next), indexed by the first component of the vertex label. Thus there exists a unique longest path from the start vertex s upto the vertex t representing that step of the sweep at which the machine either halts or makes a mode transition. (In general, this may occur after i steps, for some i less than or equal to the maximum possible length of the sweep.) To simulate the computation of the machine, this path has to be found, i.e. each vertex LAB(0)=s,LAB(1),$\cdots$,LAB(i)=t on this path has to be constructed.

The setup stage procedure construct-graph may be described in pseudo- code as follows:

```
alloc PRED:global-array of vertex-labels indexed by vertex-labels;
procedure construct-graph
seq-begin
for(each layer i)par-begin
for(each vertex v of layer i)seq-begin
if(layer(v)==0)PRED(v)←{v}; else PRED(v)←{};
if(layer(v)>0)seq-begin
for(each vertex u of layer layer(v)-1)
if(NEXT(u)==v)PRED(v)←PRED(v)U{u};
if(PRED(v)=={})PRED(v)←{adopter(layer i-1)};
seq-end
```

seq-end

par-end

/* LOOP INVARIANT MENTIONED IN THE TEXT HOLDS AT THIS POINT */

seq-end;

The path-finding procedure find-path is now presented as a multi- process algorithm. Note that the graph is acyclic and layered. One process is activated for each vertex. (The label of the vertex is itself the pid of the process and is referred to in the following pseudocode as SELF.) Each process tries to find whether its vertex is connected to the start vertex s, and it maintains a record in a global memory array PRED to represent the labels of the farthest vertices to which it knows it is connected. Let the contents of record PRED of processor i be denoted by the shorthand [i] (i.e. PRED(i) is synonymous with [i]). Let the PRED records of processors allocated to vertices of the $0^{th}$ layer (prospective initial states of the sweep) hold the value SELF (i.e. PRED(i)={i} for such i). Note that PRED(i) is a set of labels.

The initial contents of PRED(v) are all the immediate predecessors of vertex v, which can be determined by trying each vertex u of the previous layer in turn; there are only a constant number of such vertices u.) It is desired to maintain the loop invariant that all the elements in PRED(v) are from a single layer (and hence their number is bounded by a constant). However, some vertices of layers other than the $0^{th}$ layer may be orphans; i.e. they may not have predecessors in the sense given above. To handle these, introduce a chain of "adopter" vertices and set the predecessor of orphans of the $(i+1)^{th}$ layer to the "adopter" vertex of the $i^{th}$ layer. The "adopter" vertex of the $0^{th}$ layer is its own predecessor.

PRED is essentially the inverse of NEXT. However, because of the adopter vertices, the graph defined by NEXT is a subgraph of the graph defined by PRED. The algorithm works as follows:

For $O(\log((1+s(n))*(|Q|)(|\Sigma|^{wh})))=O(\log(s(n)))$ (sequential) iterations, each process executes the statement **PRED**$\leftarrow$[[**SELF**]]; (i.e. process i executes PRED(i)$\leftarrow\{k|\exists j(j\in[i]\wedge k\in[j]\})$. The global array PRED is updated by synchronising all these processes at the end of each iteration. At the end of this loop, the PRED record of a process will hold the label of the

start vertex s if and only if it is on the desired path. The array PRED is in global memory and constitutes the sole representation of the graph under construction. Processess obtain the label of the vertex they represent simply by parsing their own pid.

The path-finding stage procedure find-path may be described in pseudo- code as follows:

```
procedure find-path
seq-begin
for(log(Number-of-layers) iterations)seq-begin
/* LOOP INVARIANT MENTIONED IN THE TEXT HOLDS AT THIS POINT */
for(each vertex w)par-begin
TEMP(w)←{};
for(each vertex v∈PRED(w))for(each vertex u∈PRED(v))seq-begin
TEMP(w)←TEMP(w)U{u};
seq-end
PRED(w)←TEMP(w);
par-end
seq-end
/* LOOP INVARIANT MENTIONED IN THE TEXT HOLDS AT THIS POINT */
seq-end;
```

Once the path is obtained, the array LAB can be used as a lookup table or function. Now it is easy to obtain the new contents of all the tapes. For each location loc accessed during the sweep, knowing $mode_j$, the mode of tape j, we can calculate the largest i such that $LOC(i,mode_j)=loc$, which is the time of last access of that location. The vertex with label $LAB(i,mode_j)=<i,b,a_1,\cdots,a_j,\cdots,a_h>$ (say) represents the now known fact that just after the $i^{th}$ step of the sweep, the machine is in state b, relative to the beginning of the sweep the location of the $j^{th}$ tape head is $LOC(i,mode_j)=loc$ and the most recent w symbols read/scanned by the $j^{th}$ tape head is $a_j$, the symbol currently being scanned being $last(a_j)$. Then the subroutine DELTA when called with arguments $b,last(a_1),\cdots,last(a_j)$ returns (among other things) the final contents of the cell at location loc on tape j.

The commit stage procedure update-status may be described in pseudo- code as follows:

```
procedure update-status
seq-begin
for(each tape tapej)seq-begin
for(each location loc on tapej accessed during the sweep)par-begin
i←time of last access of location loc on tapej;
status←projection(LAB(i,modej));
newvalue←projection(DELTA(status));
tapej[headj+loc]←newvalue;
par-end;
headj←headj+(last location accessed on tapej);
modej←last w head movements of tapej;
seq-end;
state←DTM state at end of sweep;
seq-end;
```

It has already been shown in the proofs of previous lemmas that primitives like computation of head movements can be accomplished in time polylog(s(n)+n) by a single processor. It remains to be shown how the work in the algorithm described above is distributed among the processors so as to achieve the required resource bounds.

A naive implementation of the algorithm described above would use one processor per process, O(log(s(n))/log(n)) global memory words for temporaries per processor (needed in general since $s(n)=\Omega(n)$; may be omitted if $s(n)=O(poly(n))$), and O(s(n)log(s(n))/log(n)) global memory words for common data, hence O(s(n)) processors and O(s(n)log(s(n))/log(n)) global memory words in all and $t(n)=poly(x(n)+log(s(n))+log(n))$ parallel time. Since $log(s(n))=O(poly(x(n)+log(n)))$, we have $t(n)=poly(x(n)+log(n))$.

Since the number of bits per register/global memory word is O(log(n)), this means O(s(n)log(s(n))) hardware, which is too much. By using only O(s(n)/log(s(n))) processors and letting each of them (sequentially) simulate log(s(n)) processes of the algorithm above,

the amount of hardware for processors and global memory words used for temporaries can be reduced to O(s(n)) at the cost of a poly(x(n)+log(n)) factor increase in parallel time (since log(s(n))=O(poly(x(n)+log(n)))).

However, the memory requirement for common data is still too large. To reduce this, the PRED records have to be implemented using only O(1) bits. This can be done by taking advantage of the layered nature of the graph. The introduction of the adopter vertices ensures that the following loop invariant holds:

LOOP INVARIANT: The layer number of each of the elements in the PRED record of a vertex of the $i^{th}$ layer after j iterations of the loop is the same and depends only on i and j (it is equal to $\max(i-2^j,0)$)

This can be recomputed in each iteration. Hence it suffices to omit the first component of the label of the vertices which gives their layer and store only the remaining components of the label (i.e. $<b,a_1,\cdots,a_j,\cdots,a_h>$) to preserve all the information of record PRED. In each layer, there are only a finite number of vertices. Thus $O(\log((|Q|)*(|\Sigma|^{wh})))=O(1)$ bits per vertex are enough and the memory requirement reduces to O(s(n)+log(n)) bits. When the processing for a vertex is being simulated, the O(1) bits correponding to that vertex are uploaded into the global memory words used for temporaries by the simulating processor for calculations, then downloaded after the simulation of that vertex is through. So the number of bits in global memory words used for temporaries by the simulating processors is O(s(n)/log(s(n)))*O(log(s(n))/log(n))*O(log(n))=O(s(n)) which is acceptable.

∎

Because the graph is acyclic, layered and bounded-width, only O(1) hardware per vertex (O(s(n)+log(n)) for the whole graph) rather than O(1) hardware per edge ($O((s(n)+\log(n))^2)$ for the whole graph) is needed to achieve poly(x(n)+log(n)) parallel time. Thus a processor blowup similar to that faced by Pippenger [Pippenger1979] is avoided. The importance of the notion of cycle-time in the definition of mode transitions will become evident in Chapters 3 and 4, where it will be shown that a polylog(n) bound on mode transitions, though more restrictive than a corresponding bound on reversals, is still enough to obtain an analogue of

Pippenger's simulation result [Pippenger1979] for reversals, given a polynomial blowup in the workspace.

# Chapter 3

# The Calculus FO+Y(T,S,V)

In this chapter, first the calculus FO+Y(T,S,V) extending first order logic is defined. An interpretation of syntactic restrictions on formulas as resource bounds on an abstract model of computation is given. Then the expression of PRAM computations by syntactically restricted formulas is presented. The implementation of the calculus on the DTM model is covered in chapter 4.

The work reported in this chapter is an extension of Immerman's work on the expression of parallel computations and focuses on separating the representation of time instants from the representation of space locations. The price paid for this is the predominant operationality of our approach which eliminates the possibility of our calculus being used directly as a programming language. However, the purpose of this work is to prove a complexity theoretic characterisation result and we feel the use of a descriptive approach has greatly eased the task of discovering as well as presenting the proof.

## 3.1 Definition of the Calculus

Structures shall have three disjoint finite domains which shall be known as the linear domain LIN, the logarithmic domain LOG and the binary domain BIN. The number of elements in LIN $|LIN|=n$. $|LOG|=\lceil(log_2 n)\rceil$. It is assumed that $n\geq 4$. $|BIN|=2$.

Over each domain, there shall be a predefined successor function SUC that imposes a total ordering on the domain. There shall be a predefined binary equality predicate EQ over each domain. Strict typing shall be enforced and ambiguity can be resolved from the context.

W.r.t. SUC, the domains may be thought of as initial subsets of the natural numbers. The smallest element of a domain D$\in${LIN,LOG,BIN} is the predefined constant D1 and the largest element of D is the predefined constant D$\infty$. Ambiguity can be resolved from the context. SUC(D$\infty$) shall be D$\infty$ itself.

Structures shall have a predefined doubling function DOUBLE over each of the domains. DOUBLE(x)=if((x+x)$\leq$D$\infty$)then(x+x)else(D$\infty$).

Structures shall have predefined functions FROMLOG:LOG$\rightarrow$LIN and TOLOG:LIN$\rightarrow$LOG. FROMLOG(x)=x and TOLOG(x)=min(x,$\lceil(log_2 n)\rceil$).

Structures shall have predefined functions FROMBIN:BIN$\rightarrow$LIN and TOBIN:LIN$\rightarrow$BIN. FROMBIN(x)=x and TOBIN(x)=min(x,2).

In all the above, we use the interpretation of domain elements as numbers. Thus:

FROMLOG(LOG1)=LIN1;

TOLOG(LIN1)=LOG1;

TOLOG(LIN$\infty$)=LOG$\infty$;

TOBIN(LIN1)=BIN1;

FROMBIN(BIN1)=LIN1;

TOBIN(LIN$\infty$)=BIN$\infty$;

FROMBIN(BIN$\infty$)=SUC(LIN1).

Structures shall have a unary predicate INP over LIN which shall not be predefined and which shall represent a binary "input" string of length n, or the extensional database.

The *logic* FO consists of all formulas built up in the usual way from the typed symbols SUC, DOUBLE, TOLOG, FROMLOG, TOBIN, FROMBIN, EQ and INP together with logical connectives ($\wedge$, $\vee$, $\neg$), typed quantifiers ($\forall$, $\exists$), typed domain variables (x, y, z, t, s, $t_1$, $s_1$, ta, sa, tb$\cdots$) and typed predicate variables (P, Q, R$\cdots$). The predicate variables will be permitted only in suitable extensions of FO.

Concatenation of tuples and sequences shall be indicated by the symbol "$*$" which also denotes multiplication. When the same domain element is to be repeated in succession in a tuple, the number of consecutive occurrences shall be indicated as an exponent.

For a Time Domain $T\in\{LIN,LOG\}$, a Space Domain $S\in\{LIN,LOG\}$, and a Variance Domain $V\in\{LOG,BIN{:}(V\infty<S\infty)\wedge(V\infty\leq T\infty)\}$, the *calculus* FO+Y(T,S,V) is obtained by augmenting FO with the following additional formation rule:

Let $c,k,q\geq 1$ be any integers, let $\Theta$ be a formula of the calculus, let $P^{(q+k)}$ be a (q+k)-ary predicate variable over $V^qS^k$, let $\mathbf{t}$ be a c-tuple of distinct variables over T, $\mathbf{s}$ a k-tuple of distinct variables over S and $\mathbf{r}$ a q-tuple of distinct variables over V and let $\mathbf{x}$ be a (q+k)-tuple of (not necessarily distinct) constants, variables and compound terms over $V^qS^k$. Let the symbol $\Psi$ denote the formula $Y(T,S,V)[P,\mathbf{t},\mathbf{r},\mathbf{s}](\Theta)[\mathbf{x}]$.

Then $\Psi$ is a formula of the calculus. All free occurrences of the variables of $\mathbf{t}$, $\mathbf{r}$ and $\mathbf{s}$ and the predicate variable P in $\Theta$ are bound in $\Psi$ by the operator Y. The free variables of $\Psi$ are the remaining freely occurring variables of $\Theta$, as well as the variables of $\mathbf{x}$. (This also covers the free predicate variables). The calculus is closed under the usual FO connectives and quantifiers, as well as nesting of the operator Y in $\Theta$. The Y-operator constructs an interpretation, the *Y-interpretation*, of the predicate variable P, and $\Psi$ holds iff $P(\mathbf{x})$ holds in the Y-interpretation (in the context of an interpretation of the free symbols). Y computes the Y- interpretation of P by the following pseudo-algorithm:

**begin**

$\mathbf{P}\leftarrow\{\}$; /* **Empty Set** */

**for** $\mathbf{t}\leftarrow\mathbf{T1}^c$ **to** $\mathbf{T}\infty^c$ **do** /*Iteration by lexicographic ordering with $\mathbf{t}\in\mathbf{T}^c$*/

$\mathbf{P}\leftarrow\{\ \mathbf{r}*\mathbf{s}\mid \mathbf{r}*\mathbf{s}=<r_1,r_2,\cdots,r_q,s_1,s_2,\cdots,s_k>\wedge$

$\forall 1\leq j\leq q[r_j\in V]\wedge\forall 1\leq j\leq k[s_j\in S]\wedge$

$\Theta(\mathbf{P},\mathbf{t},\mathbf{r},\mathbf{s})$ **holds** $\}$;

/* **Non-monotonic update in parallel for all tuples** $\mathbf{r}*\mathbf{s}$ */

**end.**

If the enclosed formula $\Theta$ contains an occurence of the Y-operator, then the non-monotonic

update in parallel mentioned above requires a different Y-interpretation of this enclosed Y-operator for each binding of values to variables by the outer Y-operator. In other words, the enclosed Y-operator is invoked in parallel for each binding of **r** and **s**.

Occurrences of Y that are nested in the scope of another occurrence of Y are invoked over and over again at every stage (iteration) in the evaluation of the outer operator. The vocabulary of the expressions in the scope of the Y operator can vary from occurrence to occurrence, because of predicate variables bound outside its scope, though the symbols SUC, DOUBLE, TOLOG, FROMLOG, TOBIN, FROMBIN, EQ and INP are common. However, for each occurrence, the vocabulary is fixed, and the semantics of Y is defined in terms of the invocation time interpretations of the symbols of that vocabulary. This defines the semantics of Y.

Formulas of FO+Y(T,S,V) which do not contain any unbound variables can be seen to define a global predicate on finite binary strings. Thus each formula corresponds to a set of strings and the calculus expresses a class. The same descriptor shall be used to refer to the calculus and the class it expresses. Ambiguity can be resolved from the context.

The notion of "Variance Domain" introduced here generalizes simultaneous induction such that the number of predicates being simultaneously defined becomes a slowly growing function of the input size.

## 3.2 Normal Form Results for FO+Y(T,S,V)

**Definition 3.2.1:**

**The constants c, k and q (specified in the definition of the calculus) are referred to as the time arity, the space arity and the variance arity respectively of the Y-operator. The time arity (respectively space arity, variance arity) of a formula is defined to be equal to the largest among the time arities (respectively space arities, variance arities) of the Y-operators that occur in the formula. The nesting-depth of a formula is defined to be equal to the length of the longest**

chain of occurrences of Y-operators in the formula such that each successive occurrence in the chain is in the scope of the previous one. *A formula in prenex form consists of a propositional formula in the scope of a nested sequence of zero or more first order quantifiers over the linear domain LIN and Y-operators, where no Y-operator is nested immediately inside the scope of another Y-operator.*

W.l.o.g. we assume that disjunctions do not occur in formulas.

Definition 3.2.2:

The height of each negation or conjunction in a formula is the nesting depth of the subformula in its scope. The height of a Y-operator is 1 if there is another Y-operator immediately enclosed inside its scope and 0 otherwise. The height of a formula is the largest among the heights of all its negations, conjunctions and Y-operators. Note that the height of a formula in prenex form is 0. W.l.o.g., if $\Theta$ is a formula of height greater than 0, there is a unique (modulo trivial first order simplifications) leftmost innermost negation/conjunction/Y-operator in $\Theta$ whose height is equal to the height of $\Theta$ and which immediately encloses a Y-operator. The formula consisting of this negation/conjunction/Y-operator and the formula(s) inside its scope is said to be the *redex* of $\Theta$.

Claim 3.2.3:

*Let $\Theta$ be a formula not in prenex form. The height of the redex of $\Theta$ is equal to the height of $\Theta$. The height of any subformula inside the scope of the outermost operator of the redex of $\Theta$ is strictly less than the height of $\Theta$.*

Lemma 3.2.4:

*Given a formula of FO+Y(T,S,V), one can construct an equivalent formula in prenex form with the following properties: (1)The nesting depth does not increase. (2)The number of Y-operators does not increase. (3)The number of first order quantifiers does not increase.*

Proof:

By structural induction. Essentially it is shown that propositional operators can be migrated inward through Y-operators. In addition, if a Y-operator is nested immediately inside

the scope of another Y-operator, they can be replaced by a single Y-operator.

Let R be a reduction function which, given as argument a formula $\Theta$ not in prenex form, constructs formula $R(\Theta)$ by replacing the redex of $\Theta$ with a formula which is equivalent to the redex of $\Theta$ and whose height is strictly less than the height of the redex of $\Theta$, without increasing the nesting depth, the number of Y-operators or the number of first-order quantifiers. Then by repeated applications of R and trivial first order simplifications, one can construct from formula $\Theta$ a formula $\Phi$ which is equivalent to $\Theta$, which is in prenex form and which satisfies the properties listed in the statement of the lemma. W.l.o.g. the redex of a formula $\Theta$ not in prenex form is in one of the following forms:

1. $\neg(Y(T,S,V)[P,\mathbf{t},\mathbf{r},\mathbf{s}](G)[\mathbf{x}])$
2. $F \wedge Y(T,S,V)[P,\mathbf{t},\mathbf{r},\mathbf{s}](G)[\mathbf{x}]$
3. $Y(T,S,V)[P,\mathbf{t},\mathbf{r},\mathbf{s}](H)[\mathbf{y}] \wedge Y(T,S,V)[P,\mathbf{t},\mathbf{r},\mathbf{s}](G)[\mathbf{x}]$
4. $Y(T,S,V)[P,\mathbf{ta},\mathbf{ra},\mathbf{sa}](Y(T,S,V)[Q,\mathbf{tb},\mathbf{rb},\mathbf{sb}](G)[\mathbf{x}])[\mathbf{y}]$

where F, G and H are formulas and the outermost operator of F is not a Y-operator. It is sufficient to handle these forms to obtain a suitable reduction function R.

Let a redex be in form 1. Increasing the arity of the time domain from c to c+1 (thus introducing a new time variable ta), construct a single formula $Y(T,S,V)[P,\mathbf{ta}*\mathbf{t},\mathbf{r},\mathbf{s}](I)[\mathbf{x}]$ with the following formula I:

$[\mathbf{ta}=T1 \wedge G] \vee [\mathbf{ta}*\mathbf{t}=T\infty^{c+1} \wedge \neg P(\mathbf{rs})] \vee$

$[\mathbf{ta}\neq T1 \wedge \mathbf{ta}*\mathbf{t}\neq T\infty^{c+1} \wedge P(\mathbf{rs})]$

The resulting formula is equivalent to the redex and its height is less by 1.

Let a redex be in form 2. Rename the variables P, t, r and s bound by the Y-operator to avoid conflicts. This changes G to G′ (say). Increasing the arity of the time domain from c to c+1 (thus introducing a new time variable ta), construct a single formula $Y(T,S,V)[P,\mathbf{ta}*\mathbf{t},\mathbf{r},\mathbf{s}](I)[\mathbf{x}]$ with the following formula I: (note that the P, t, r and s bound by the Y-operator in the formula under construction represent the renamed variables)

$[\mathbf{ta}=T1 \wedge G'] \vee [\mathbf{ta}\neq T1 \wedge F \wedge P(\mathbf{rs})]$.

The resulting formula is equivalent to the redex and its height is less by 1.

Let a redex be in form 3. A time domain arity of c=max(c1,c2)+1, a space domain arity of k=max(k1,k2) and a variance domain arity of q=max(q1,q2)+1 are used. By introducing conjuncts tying down some of the bound variables to fixed values, one can ensure that computation progresses only for the required time. The approach is to compute the two Y-interpretations in separate workspaces, query these on **y** and **x**, and store the conjunction of the truth values so obtained in a third workspace. These workspaces are discriminated by the most significant bound variable of r. Construct a single formula $Y(T,S,V)[P,ta*t,ra*r,s](I)[S\infty^{(q+k)}]$ with the following formula I: (ta and **t** are 1- and (c-1)-tuples of variables over T, ra and **r** are 1- and (q-1)-tuples of variables over V and **s** is a k-tuple of variables over S)

$[ta=T1 \wedge ([ra=V1 \wedge H'(t,r,s)] \vee [ra=SUC(V1) \wedge G'(t,r,s)])] \vee$

$[ta*t=T\infty^c \wedge P(V1*y') \wedge P(SUC[V1]*x')] \vee$

$[ta \neq T1 \wedge ta*t \neq T\infty^c \wedge P(ra*r*s)]$

where H′ and G′ are obtained from H and G by introducing conjuncts that tie down unused variables to D1 and **y**′ and **x**′ are obtained from **y** and **x** by prefixing enough occurrences of D1 to match the required arity. This resulting formula is equivalent to the original redex and its height is less by 1.

Let a redex be in form 4. A time domain arity of c=c1+c2+2, a space domain arity of k=k1+k2 and a variance domain arity of q=q1+q2+1 are used. By introducing conjuncts tying down some of the bound variables to fixed values, one can ensure that computation progresses only for the required time. The approach is to store the Y-interpretations in separate workspaces during the nested computations. There is one outer Y-interpretation and $V\infty^{q1}S\infty^{k1}$ inner Y-interpretations, one for each binding of the variables of **ra** and **sa**. The inner Y-interpretations are evaluated on **x** in every stage of the computation of the outer Y-interpretation. When the outer Y-interpretation has been computed, it is evaluated on **y** and the truth value obtained is stored in a third workspace. These workspaces are discriminated by the most significant bound variable of the variance domain.

Let x=xv*xs and y=yv*ys where xv and yv are q2- and q1-tuples of terms over V and xs

and **ys** are k2- and k1-tuples of terms over S. Construct a single formula Y(T,S,V)[R,tc*ta*td*tb,r with the following formula I: (tc, **ta**, td and **tb** are 1-, c1-, 1- and c2-tuples of variables over T, rc, **ra** and **rb** are 1-, q1- and q2-tuples of variables over V and **sa** and **sb** are k1- and k2-tuples of variables over S)

[tc=T1 ∧ td=T1 ∧ ([rc=V1 ∧ R(rc*ra*rb*sa*sb)] ∨ [rc=SUC(V1) ∧ G′])] ∨

[tc=T1 ∧ td*tb=SUC(T1)$^{c2+1}$ ∧ (rc=V1 ∧ R[SUC(V1)*ra*xv*sa*xs])] ∨

[tc=T1 ∧ td≠T1 ∧ td*tb≠SUC(T1)$^{c2+1}$ ∧ R(rc*ra*rb*sa*sb)] ∨

[tc*ta*td*tb=SUC(T1)$^{c1+c2+2}$ ∧ R(V1*yv*rb*ys*sb)] ∨

[tc≠T1 ∧ tc*ta*td*tb≠SUC(T1)$^{c1+c2+2}$ ∧ R(rc*ra*rb*sa*sb)]

where G′ is obtained from G by introducing conjuncts that tie down unused variables to D1. This resulting formula is equivalent to the original redex and its height is less by 1. The principle here is that all "computation" takes place while tc=T1 (the first three disjuncts), except for the test of the outer predicate on the tuple **y**, which is done when the variables of tc, **ta**, td and **tb** are all equal to SUC(T1) (the fourth disjunct). The last (fifth) disjunct says that at all other times the status quo is preserved. The first three disjuncts follow a similar principle for the computation of the inner Y-operator.

These constructions together give a reduction function R. By induction alternately on the number of negations/conjunctions whose height is equal to the height of the formula while there is at least one such negation/conjunction and then on the number of Y-operators whose height is non-zero, it follows that each formula is constructively equivalent to some formula in prenex form. It suffices to note that the current induction variable strictly decreases in each step and induction variables of yet-to-come steps are not increased by more than a finite amount in the current step. This completes the proof of the lemma.

∎

**Definition 3.2.5:**

**A formula is said to be in collapsed form if it consists of a propositional formula in the scope of at most a single Y-operator.**

Note that a formula in collapsed form has no first order quantifiers in it.

**Lemma 3.2.6:**

*The formulas of the calculi FO+Y(LIN,LIN,V), FO+Y(LOG,LIN,V) and FO+Y(LIN,LOG,BIN), where V∈{LOG,BIN}, have a collapsed form, i.e. given a formula of one of these calculi, one can construct an equivalent formula in collapsed form.*

**Proof:**

Note that the algorithm in the proof of the previous lemma which converts a formula into prenex form does not ever increase the number of quantifiers or Y-operators or the nesting depth. Thus if a way were to be found of simulating quantifiers over the linear domain LIN using Y-operators, negations and conjunctions, it would be possible to attain a collapsed form. (Nonessential use of disjunctions is made below for clarity.)

The following formula gives such a simulation when the space domain is LIN, i.e. in the case of FO+Y(LIN,LIN,V) and FO+Y(LOG,LIN,V):(V∈{BIN,LOG})

Y(T,S,V)[P,t,r,s](F′)[S1] simulates ∀sF(s) for some formula F with F′ as

[t=T1 ∧ F(s)] ∨ [t≠T1 ∧ t≠T∞ ∧ P(s) ∧ P(DOUBLE(s)) ∧ P(SUC(DOUBLE(s)))] ∨

[t=T∞ ∧ P(S1) ∧ P(SUC(S1)) ∧ P(SUC(SUC(S1))) ∧ P(SUC(SUC(SUC(S1))))]

The following formula gives such a simulation when the time domain is LIN, i.e. in the case of FO+Y(LIN,LIN,V) and FO+Y(LIN,LOG,BIN):(V∈{BIN,LOG})

Y(T,S,V)[P,t,r,s](F′)[S1] simulates ∀tF(t) for some formula F with F′ as

[t=T1 ∧ F(t)] ∨ [t≠T1 ∧ P(S1) ∧ F(t)]

Existential quantifiers are handled similarly. The collapsed form is obtained by first successively simulating all occurrences of first-order quantifiers in the manner described, and then taking the prenex form as per the previous lemma. Since there are no first-order quantifiers left and none are added while taking the prenex form, and since immediately nested Y-operators are not allowed in the prenex form, it follows that the formula must be in collapsed form.

∎

It follows that formulas of FO+Y(LIN,LIN,V), FO+Y(LOG,LIN,V) and FO+Y(LIN,LOG,V)

(V∈{BIN,LOG}) have a collapsed form. We shall call these as collapsible calculi.

**Corollary 3.2.7:**

*Restricting the nesting depth to 1 does not affect the expressive power of collapsible calculi if the time domain arity, the variance domain arity and the space domain arity are unrestricted.*

**Lemma 3.2.8:**

*Restricting the time domain arity to 1 does not affect the expressive power of FO+Y(T,S,V) when nesting is permitted.*

**Proof:**

Let Y[P,t,r,s](F)[x] be some formula with time-arity c, variance arity q and space-arity k. Let P1, P2,···,Pc be c new predicate variables over $V^qS^k$. Then the following formula is equivalent to the given formula and its time-arity is restricted to 1:

Y[P1,t1,r,s](Y[P2,t2,r,s](···(Y[Pc,tc,r,s](F′)[r∗s])···)[r∗s])[x]

where F′ is

[(t2=T1 ∧ t3=T1 ∧ ··· ∧ tc=T1 ∧ F[P1/P]) ∨

(¬(t2=T1) ∧ t3=T1 ∧ ··· ∧ tc=T1 ∧ F[P2/P]) ∨

(¬(t3=T1) ∧ t4=T1 ∧ ··· ∧ tc=T1 ∧ F[P3/P]) ∨

( ··· ) ∨ (¬(tc=T1) ∧ F[Pc/P])

]

and F[Q/P] denotes the replacement of all unbound occurences of predicate symbol P in formula F with the predicate symbol Q.

t1, t2, etc. are the variables of the c-tuple t. No renaming of domain variables is necessary. The principle here is that the last stored value of the predicate P is at the innermost Y-operator which is not on the first iteration of the current invocation. The only exception is the first step of the entire computation, when we use the P1 ← {} (Empty Set) initialisation to start up. Note that the space arity is not affected.

∎

**Corollary 3.2.9:**

*Every formula of a collapsible calculus has a form which is a propositional formula in the*

*scope of a sequence of immediately nested time-monadic Y-operators.*

In the rest of this study, only the collapsible calculi are considered.

The following definition and lemma of technical interest is used in Chapter 4:

**Definition 3.2.10:**

**A formula Y(T,S,V)[P,t,r,s](F)[w] is said to be in standard form if it is in collapsed form with the additional property that there is no occurrence of the functions SUC and DOUBLE over the LOG and BIN domains in the formula and in any (q+k)-tuple x which is an argument of an occurrence of P in F, the first q terms of the tuple (which in general may be constructed from any constant or variable and the predefined functions, but which take values from the variance domain) do not contain any occurrence of the variables of s.**

**Lemma 3.2.11:**

*Given any formula Y(T,S,V)[P,t,r,s](F)[w] in collapsed form, with time arity c, space arity k and variance arity q, one can construct an equivalent formula Y(T,S,V)[P,t',r,s](F')[w] in standard form, with time arity c', space arity k (unchanged) and variance arity q (unchanged).*

**Proof:**

Occurences of the functions SUC and DOUBLE over the domains LOG and BIN are handled by functional composition from those over the domain LIN, together with the functions FTLOG(.)=FROMLOG(TOLOG(.)) and FTBIN(.)=FROMBIN(TOBIN(.)). This is done using the following identities repeatedly as (left-to-right) rewrite rules:

TOLOG(FROMLOG(x))=x

TOBIN(FROMBIN(x))=x

TOBIN(FROMLOG(TOLOG(x)))=TOBIN(x)

FROMLOG(TOLOG(FROMBIN(x)))=FROMBIN(x)

$SUC_{LOG}$(x)=TOLOG($SUC_{LIN}$(FROMLOG(x)))

$SUC_{BIN}$(x)=TOBIN($SUC_{LIN}$(FROMBIN(x)))

$DOUBLE_{LOG}$(x)=TOLOG($DOUBLE_{LIN}$(FROMLOG(x)))

$DOUBLE_{BIN}(x)$=TOBIN($DOUBLE_{LIN}$(FROMBIN(x)))

where the subscript of SUC and DOUBLE is used here to indicate the domain of these strictly typed functions. The identities enable SUC and DOUBLE to be simulated over the LIN domain by standard algorithms given in the next chapter. When the innermost constant or variable of a term belongs to domain LIN, the typing rules ensure that the resulting term is expressed in terms of SUC and DOUBLE over the domain LIN together with FTLOG(.)= FROMLOG(TOLOG(.)) and FTBIN(.)= FROMBIN(TOBIN(.)) for which standard simulation algorithms are given in the next chapter.

Each occurrence of a space domain variable that violates the second requirement is eliminated in succession at the expense of increasing the time arity by 1. The calculus permits F.O. quantifiers over the variance domain. If a variance domain term $\mathbf{y}$ containing a space domain variable is present in the argument $\mathbf{x}$ of an occurrence $P(\mathbf{x})$ of the predicate variable P in the formula, replace the occurrence $P(\mathbf{x})$ by the expression $\exists r_0[P(\mathbf{x}[r_0/\mathbf{y}])\wedge EQ(r_0,\mathbf{y})]$, for some new variance domain variable $r_0$. The existential quantifier is then migrated outward to get a prenex form $Y(T,S,V)[P,\mathbf{t},\mathbf{r},\mathbf{s}](\exists r_0 F'')[\mathbf{w}]$.

Finally the quantifier is simulated as follows (the method of the previous lemmas cannot be used since it reintroduces a space domain variable in the variance domain term): The time arity of the Y-operator is increased by 1, adding a new (least significant) time domain variable $\mathbf{t0}$. Thus $\mathbf{t}'=\mathbf{t}t_0$ and $c'=c+1$. Let TIM2VAR denote the appropriate predefined function taking time domain arguments to variance domain values. ($T\infty \geq V\infty$ by assumption). Let $F'$ denote $(P(\mathbf{rs}) \vee F''[TIM2VAR(t_0)/r_0])$. The formula $Y(T,S,V)[P,\mathbf{t}',\mathbf{r},\mathbf{s}](F')[\mathbf{w}]$ is equivalent to the original formula and satisfies the requirements of the lemma.

∎

**Theorem 3.2.12:**

*Given any formula of the calculus FO+Y(T,S,V) (such that T and S are not both LOG), one can construct an equivalent formula in standard form.*

**Proof:**

Follows from the previous results.

## 3.3 Logical Expression of PRAM Computations

In chapter 2, the sequential and parallel models of computation and the resources of interest were defined. Then the resource bounded simulation of the DTM model on the PRAM model was presented. In the previous section, the calculus FO+Y(T,S,V) extending first order logic was defined and a standard form was demonstrated. In this section, the simulation of the PRAM model in the calculus is presented. The simulation of the logic on the DTM model is presented in chapter 4.

We use the techniques of descriptive complexity research to structure the proof of our result simulating PRAMs on DTMs. While it appears to us very difficult to present the direct simulation of the PRAM on the DTM, the collapsible calculi FO+Y(T,S,V) used as a via media turn out to be both powerful enough to express PRAM computations as well as (syntactically) simple enough to permit easily presented simulations on the DTM. Thus we can use the Y-operator indiscriminately while writing a formula expressing the PRAM computation, and then construct the equivalent formula in standard form before simulating it on the DTM. The standard form gives a regular structure to the "global memory access pattern" of PRAM algorithms and the normalised algorithm represented by the expression in standard form can be implemented efficiently in terms mode transitions on the DTM, with a polynomial blowup in the workspace. The proof is a variant of Immerman's proof that the class CRAM[t(n)]-PROC[O($n^k$)] is included in the class IND[t(n)]-VAR[2k+2] [Immerman1987b].

For the purposes of this section, *conventional hardware bounds* h(n) and *conventional parallel time bounds* t(n) are those of the form O($\log(n)^k$), O($n^k$) and O*($n^k$) (for some constant $k \geq 1$), with the restriction that either $t(n)=\Omega(n)$ and $h(n)=O(poly(t(n)))$ or $h(n)=\Omega(n)$ and $\log(h(n))=O(poly(t(n)+\log(n)))$.

Each processor has a finite set of registers including the PROCESSOR, ADDRESS, CONTENTS and PROGRAM_COUNTER registers.

**Claim 3.3.1:**

*Let h(n) be a conventional hardware bound and let t(n) be a conventional parallel time bound. Then, for suitable choice of domains T, S and V, and corresponding domain arities c, k and q respectively, the following hold:*

*(1) Time instants can be represented as elements of* $T^c$.

*(2) Addresses of bits in processor registers and global memory can be represented as elements of* $V^q S^k$.

*(3)* $T\infty^c = O(poly(t(n)+log(n)))$.

*(4)* $V\infty^q * S\infty^k = O(h(n))$.

**Theorem 3.3.2:**

*For* $T,S \in \{LIN,LOG\}$ *(such that T and S are not both LOG) and* $V \in \{LOG,BIN: V\infty < S\infty \wedge V\infty \leq T\infty\}$, *let A be a CRCW PRAM decision algorithm written using only READ, WRITE, ADD, SUBTRACT, MOVE, INC/DEC, BLT and HALT instructions, which operates with* $O(poly(V\infty * S\infty))$ *hardware and* $poly(T\infty)$ *parallel time. Then, there exists a formula of FO+Y(T,S,V) which expresses the language decided by A.*

**Proof:**

First we show how the total ordering and BIT predicates can be expressed. The expressions are given only for FO+Y(LIN,LOG,BIN) and FO+Y(LOG,LIN,BIN), where each of these calculi is restricted to space arity 1. These are the tightest restrictions, and from these, the expressions for all calculi FO+Y(T,S,V) (T,S∈{LOG,LIN};V∞<S∞;T,S not both LOG) with and without restriction of space arity, can be obtained easily.

The total ordering predicate ≤ on domain LIN is defined in FO+Y(LIN,LOG,BIN) as (note that the variance arity is zero)

≤(x,y)≡Y(P,t,,s)[F](LOG∞)

where

F≡[(EQ(s,LOG1) ∧ EQ(t,x)) ∨

(EQ(s,LOG∞) ∧ EQ(t,y) ∧ P(LOG1)) ∨

EQ(x,y) ∨

P(s)

]

and in FO+Y(LOG,LIN,BIN) as

$\leq(x,y) \equiv Y(P,t,r,s)[F](3*LIN\infty)$

where

$F \equiv [(EQ(t,LOG1) \wedge (r=1) \wedge EQ(s,x)) \vee$

$(EQ(t,LOG1) \wedge (r=2) \wedge EQ(s,y)) \vee$

$(\neg EQ(t,LOG1) \wedge (r=1) \wedge EQ(x,DOUBLE(s))) \vee$

$(\neg EQ(t,LOG1) \wedge (r=1) \wedge EQ(x,SUC(DOUBLE(s)))) \vee$

$(\neg EQ(t,LOG1) \wedge (r=2) \wedge EQ(y,DOUBLE(s))) \vee$

$(\neg EQ(t,LOG1) \wedge (r=2) \wedge EQ(y,SUC(DOUBLE(s)))) \vee$

$((r=3) \wedge EQ(x,y)) \vee$

$((r=3) \wedge \exists z(P(1*z) \wedge P(2*SUC(z)))) \vee$

$((r=3) \wedge \exists z(P(1*z) \wedge P(2*SUC(SUC(z))))) \vee$

$((r=3) \wedge \exists z(P(1*z) \wedge P(2*SUC(SUC(SUC(z)))))) \vee$

$((r=3) \wedge P(3*s))$

]

Total ordering in other domains is easily expressed in terms of total ordering on LIN and the predefined functions. In what follows, we use the symbol "$\leq$" to denote total ordering in all domains.

The following predicates are FO-definable:

$ODD'(x) \equiv \neg\exists y[EQ(x,DOUBLE(y))]$

$EVEN'(x) \equiv (\exists y \forall z)[EQ(x,DOUBLE(y)) \wedge$

$\neg(\neg EQ(y,z) \wedge EQ(y,SUC(z)) \wedge EQ(x,DOUBLE(z)))$

]

$MAX(x) \equiv EQ(x,SUC(x))$

$ODDMAX(x) \equiv [MAX(x) \wedge (\exists y)[\neg EQ(x,y) \wedge EQ(x,SUC(y)) \wedge EVEN'(y)]]$

$EVENMAX(x) \equiv [MAX(x) \wedge (\exists y)[\neg EQ(x,y) \wedge EQ(x,SUC(y)) \wedge ODD'(y)]]$

$ODD(x) \equiv [ODD'(x) \vee ODDMAX(x)]$

$EVEN(x) \equiv [EVEN'(x) \vee EVENMAX(x)]$

$HALF'(x,y) \equiv [\neg EQ(x,y) \wedge [EQ(x,DOUBLE(y)) \vee EQ(x,SUC(DOUBLE(y)))]]$

$HALF(x,y) \equiv [HALF'(x,y) \wedge \neg(\exists z)[\neg EQ(y,z) \wedge EQ(y,SUC(z)) \wedge HALF'(x,z)]]$

The BIT predicate is implicitly FO-definable as follows:

$BIT'(x,n) \equiv [[(EQ(n,LIN1) \wedge ODD(x)) \vee$

$(\exists y,m)[HALF(x,y) \wedge \neg EQ(n,m) \wedge EQ(n,SUC(m)) \wedge BIT'(y,m)]] \wedge$

$(\forall y,z)[EQ(y,z) \vee$

$(\exists m)[(BIT'(y,m) \wedge \neg BIT'(z,m)) \vee$

$(BIT'(z,m) \wedge \neg BIT'(y,m))$

$]$

$]$

$]$

$BIT(x,n) \equiv (\exists y)[\neg EQ(x,y) \wedge EQ(x,SUC(y)) \wedge BIT'(y,n)]$

The definition of the BIT predicate in terms of the BIT′ predicate is FO. It remains to give the explicit definition of the BIT′ predicate in the extended calculi.

In FO+Y(LOG,LIN,BIN), BIT′ can be defined as follows:

$BIT'(x,n) \equiv Y(P,t,r,s)[F](3*LIN\infty)$

where

$F \equiv [(EQ(t,LIN1) \wedge [EQ(n,LIN1) \wedge ODD(x)] \wedge (r=3) \wedge EQ(s,LIN\infty)) \vee$

$(EQ(t,LIN1) \wedge \neg[EQ(n,LIN1) \wedge ODD(x)] \wedge$

$[((r=1) \wedge HALF(x,s)) \vee$

$((r=2) \wedge \neg EQ(n,s) \wedge EQ(n,SUC(s)))$

$]) \vee$

$(P(3*LIN\infty) \wedge (r=3) \wedge EQ(s,LIN\infty)) \vee$

$(\neg EQ(t,LIN1) \wedge$

$[P(2*LIN1) \wedge (\exists y)(P(1*y) \wedge ODD(y))] \wedge$

$(r=3) \wedge EQ(s,LIN\infty)) \vee$

$((\exists x',n')[P(1*x') \wedge P(2*n') \wedge \neg(EQ(n',LIN1) \wedge ODD(x')) \wedge$

$[((r=1) \wedge HALF(x',s)) \vee ((r=2) \wedge \neg EQ(n',s) \wedge EQ(n',SUC(s)))]$

$])$

$]$

In FO+Y(LIN,LOG,BIN), BIT′ can be directly defined as follows:

$BIT'(x,n) \equiv (n \leq TOLIN(LOG\infty)) \wedge Y(P,t,r,s)[F](2*TOLOG(n))$

where

$F \equiv [((r=1) \wedge [(\neg P(1*s) \wedge (\forall z<s)P(1*z)) \vee (P(1*s) \wedge (\exists z<s)\neg P(1*z))]) \vee$

$((r=2) \wedge EQ(t,x) \wedge P(1*s)) \vee ((r=2) \wedge P(2*s))]$

With $\leq$ and BIT extending FO, addition and subtraction are known to be expressible [StockmeyerVishkin1984] [Immerman1987b]. We now show how to simulate the computation of a PRAM M. On input x, |x|=n, M runs in t(n) synchronous steps, using p(n) processors and m(n) global memory cells. We consider only the cases (t(n)=polylog(n) and p(n),m(n)=$O(n^k)$, $k \geq 1$) and (t(n)=poly(n) and p(n),m(n)=$O((\log(n))^k)$,$k \geq 1$). Since the number of processors and global memory cells are bounded by some polynomial in n (log(n) in the second case), we need only a constant number of variables over the LIN domain (LOG domain in the second case) and the BIN domain to name any processor or global memory cell. Since the number of time steps and the wordwidth of registers and global memory cells are bounded by some polynomial in log(n) (n in the second case), we need only a constant number of variables over the LOG domain (LIN domain in the second case) to name any time instant or bit within a given word. Further, the number of distinct names so created are O(p(n)+m(n)) and poly(t(n)+width(n)) respectively. We can thus define the contents of all the relevant registers and global memory cells at any time instant, while keeping the number of variables over LIN (LOG in the second case) in check.

Say VALUE(loc,tim,pos,bool) has the meaning that the bit in position pos at location loc just after step tim is bool. It is straightforward to write a FO+Y(LOG,LIN,BIN) (FO+Y(LIN,LOG,BIN) in the second case) expression for VALUE if the PRAM instruc-

tions are expressed. By explicitly using the iterative and non-monotonic update abilities of the Y-operator for READ, WRITE and MOVE, searching for the last time step at which the cell was written to, and the lowest numbered processor which attempted a WRITE at that time, becomes unnecessary.

It is clear that the initial state can be expressed, using BIT to express the fact that the initial contents of each processor's PROCESSOR register is its processor number [Immerman1987b]. Addition and subtraction have already been mentioned. It remains to show that BLT (Branch-if-less-than- zero) and HALT are expressible. This follows since the contents of the register PROGRAM_COUNTER are available as data objects, and the program is of finite length.

∎

# Chapter 4

# Relational transformations

In chapter 2, the sequential and parallel models of computation and the resources of interest were defined. Then the resource bounded simulation of the DTM model on the PRAM model was presented. In chapter 3, the calculus FO+Y(T,S,V) extending first order logic was defined and a standard form was demonstrated. Then the simulation of the PRAM model in the calculus was presented. The evaluation of formulas in standard form on the DTM model is presented in this chapter.

The standard form gives a regular structure to the "global memory access pattern" of PRAM algorithms and the normalised algorithm represented by the expression in standard form can be implemented "efficiently" in terms of mode transitions on the DTM (a polynomial blowup of the workspace occurs while expressing the algorithm in the calculus). The following properties of a formula Y(T,S,V)[P,t,r,s](F)[w] in standard form are used:

(1) There are no first-order quantifiers and there is at most one Y-operator, which is at the outermost level. This means that the predicate symbol bound by the Y-operator can be viewed as the workspace of the DTM computation, and the time arity can be viewed as the iteration count of a simple loop.

(2) The subformula F is propositional. This means that once the truth values of the atomic predicates are known, the subformula F can be evaluated for each given binding in

O(1) time by the delta function of the DTM.

(3) In any (q+k)-tuple x which is an argument of an occurrence of P in F, the first q terms of the tuple (which in general may be constructed from any constant or variable and the predefined functions, but which take values from the variance domain) do not contain any occurrence of the variables of s. This means that the new binding of the predicate P can be constructed in $V\infty^q$ sequential iterations corresponding to the first q terms of the tuple, with each iteration involving the computation of a map from $S^k$ to $S^k$.

(4) There is no occurrence of the functions SUC and DOUBLE over the LOG and BIN domains in the formula. Since all occurrences of the functions SUC and DOUBLE take arguments only from the domain LIN, deep nesting of the typecasting functions TOBIN, FROMBIN, TOLOG and FROMLOG can be collapsed. The map from $S^k$ to $S^k$ mentioned above can be expressed entirely in terms of variables and functions over the LIN domain.

Note that k-ary predicates over the LIN domain can be viewed as binary strings of length $n^k$ stored on a DTM tape. A k-tuple of variables over the LIN domain can be viewed as a position of the tape head. A predefined function like SUC can be viewed as a map from head positions to head positions. The key idea of this chapter is that k-ary predicates can also be viewed as the contents of a one-bit register in a $n^k$-processor fixed connection network. A k-tuple of variables over the LIN domain can be viewed as a processor index. A predefined function like SUC can be viewed as a data transformation to be achieved by routing messages over the network using the properties of the interconnection functions. The way this key idea is applied here is by interpreting familiar interconnection functions as functions over binary strings and showing that these are efficiently implementable on the DTM.

## 4.1 Basic Redistributive Functions

One requirement of parallel architectures that does not apply to serial architectures is the necessity for rearranging data in order to avoid memory access contention while providing fast parallel access. A good parallel architecture design is highly dependent on how efficiently

the data manipulating functions required by the intended applications are implemented. The circuits used to achieve these functions can be considered to form an independent functional block, called a data manipulator (or interconnection) network.

To rearrange data in shared memory, or, equivalently, to redistribute data over a network, a programmed sequence of basic functions must be executed. The choice of basic functions to be hardwired determines the set of data manipulations that are achievable as well as their efficiency. Several authors have considered such basic function sets, commonly including functions from the following four categories:

1 Bit-Permute-Complement (BPC) permutation [Fraser1976] [NassimiSahni1981b] [YewLawrie1981] [NassimiSahni1982]

2 p-ordering and cyclic shift within segments [Lawrie1975] [Lenfant1978] [YewLawrie1981] [NassimiSahni1981b]

3 broadcast [Feng1974] [Thompson1978] [Siegel1979] [Parker1980] [NassimiSahni1981a]

4 masking [Feng1974] [Siegel1977] [Siegel1979]

In simulating FO+Y(LOG,LIN,V) formulas on the multitape DTM model of computation, it will prove necessary to rearrange the tape contents to reduce the number of mode transitions. Pippenger [Pippenger1979] proposed to use sorting network techniques [Batcher1968] [Stone1971] for reducing the number of reversals. We, of course, have the additional requirement of ensuring that the workspace used is optimal within constant factors.

We define a set of data manipulation functions M and a set of housekeeping functions H and show that they can be computed on multitape DTMs using O(n) workspace and polylog(n) mode transitions on inputs of length n. ($\Sigma$=$\{0,1\}$) (We give actual constructions in pseudocode for each function.) The proof of the simulation result makes extensive use of these functions.

Some of the function names have subscripts which are integers. These subscripts are additional inputs to the corresponding functions. We use the following encoding of integers as strings: The empty string represents the integer 0. Strings from the set $\{1\}^+$ represent positive integers. Strings from the set $\{0\}^+$ represent negative integers. Other strings are

not interpreted as numbers. Integers encoded as above are also interpreted as strings.

Let a string $x \in \Sigma^*$, $|x|=2^m$, be laid out on a DTM tape. The leftmost symbol of x is said to have address 0, and, incrementing to the right, the rightmost symbol has address $(2^m-1)$. A *data manipulation function* $f \in M$ is specified by declaring that the symbol at address i in the string f(x) is the same as the symbol at address f'(i) in string x, for some function f' on integers whose domain is $[0-(2^m-1)]$ and range is a subset of $[0-(2^m-1)]$. We refer to the function f' as an *address transformation* and to the function f as the corresponding *data transformation.* (The prime ' distinguishes an address transformation from the corresponding data transformation.) We refer to the binary expansion of address i as $[i_{m-1}i_{m-2}\cdots i_1 i_0]$ where $i=i_{m-1}2^{m-1}+i_{m-2}2^{m-2}+\cdots+i_1 2^1+i_0 2^0$.

The set M contains the following functions:

1 Outer Shuffle $so_m$:

The address transformation is

$so'_m([i_{m-1}i_{m-2}\cdots i_1 i_0])=[i_{m-2}\cdots i_1 i_0 i_{m-1}]$

or equivalently,

$so'_m(i)=2i$ if $0 \le i \le (2^{m-1}-1)$

$2i+1-2^m$ if $2^{m-1} \le i \le (2^m-1)$

2 Inner Shuffle $si_m$:

The address transformation is

$si'_m([i_{m-1}i_{m-2}\cdots i_1 i_0])=[i_{m-1}i_{m-2}\cdots i_0 i_1]$

or equivalently,

$si'_m(i)=i$ if $(i \bmod 4) \in \{0,3\}$

$4*\lfloor (i/4) \rfloor+3-(i \bmod 4)$ if $2^{m-1} \le i \le (2^m-1)$

3 Skew $sk_{c,m}$, for each $2 \le c \le m$:

The address transformation is

$sk'_{c,m}([i_{m-1}i_{m-2}\cdots i_{c+1}i_c i_{c-1}\cdots i_1 i_0])=[i_{m-1}i_{m-2}\cdots i_{c+1}i_c i_0\cdots i_0 i_0]$

or equivalently,

$sk'_{c,m}(i)=2^c*\lfloor (i/2^c) \rfloor+(2^c-1)*(i \bmod 2)$

The last c bits of the address are replaced by the last bit extended c times.

4 Select $selc_m$, for each $c \in \Sigma$, i.e. $sel0_m$ and $sel1_m$:

The address transformation is

$selc'_m([i_{m-1}i_{m-2}\cdots i_1 i_0])=[i_{m-1}i_{m-2}\cdots i_1 c]$

or equivalently,

$selc'_m(i)=2*\lfloor(i/2)\rfloor+c$

The last bit of the address is replaced by c.

5 Successor $suc_{c,m}$, for each $2 \leq c \leq m$:

The address transformation is

$suc'_{c,m}(i)=2^c*\lfloor(i/2^c)\rfloor+\min((2^c-1),((i \bmod 2^c)+1))$

or equivalently,

$suc'_{c,m}(i)=$ i+1 if $(i \bmod 2^c)<(2^c-1)$

i if $(i \bmod 2^c)=(2^c-1)$

Increment if there is no carry out of the $c^{th}$ bit (i.e. saturating increment of the last c bits).

6 Double $db_{c,m}$, for each $2 \leq c \leq m$:

The address transformation is

$db'_{c,m}(i)=2^c*\lfloor(i/2^c)\rfloor+\min((2^c-1),(2(i \bmod 2^c)+1))$

or equivalently,

$db'_{c,m}(i)=2^c*\lfloor(i/2^c)\rfloor+(2(i \bmod 2^c)+1)$ if $(2(i \bmod 2^c)+1)<2^c$

$2^c*\lfloor(i/2^c)\rfloor+(2^c-1)$ if $(2(i \bmod 2^c)+1)\geq 2^c$

Saturating double of the last c bits. This definition may appear nonintuitive. This is because the interpretation of domain elements as numbers that was chosen for the logic mapped the LIN domain to the range [1- n] while the addresses are in the range [0-(n-1)]. We have defined the function this way to simplify the simulation of the logic.

7 Log Truncate $trlog_{c,m}$, for each $2 \leq c \leq m$:

The address transformation is

$trlog'_{c,m}(i)=2^c*\lfloor(i/2^c)\rfloor+(c-1)$ if $(i \bmod 2^c)\geq(c-1)$

i if (imod$2^c$)$\leq$(c-1)

8 Const Truncate $tr2_{c,m}$, for each $2 \leq c \leq m$:

The address transformation is

$tr2'_{c,m}(i) = 2^c * \lfloor (i/2^c) \rfloor + 1$ if (imod$2^c$)$\geq 1$

i if (imod$2^c$)$\leq 1$

The functions in M are defined in terms of address transformations, so the length of the output string is the same as the length of the input string. Also these functions are computed by circuits with no gates: each output bit is the image of a particular bit of the input. Thus they may be described as non-inverting, length preserving projections.

Apart from these, we shall also need some functions that take more than one argument and some functions whose output string differ in length from that of the input. We describe below the set H of such functions. The functions in H perform string manipulations and arithmetic computations.

The set H contains the following functions:

1 zero: This is a function with no argument which returns the integer 0.

2 dec: On input integer n, returns the integer n-1.

3 inc: On input integer n, returns the integer n+1.

4 half: On input integer n, returns the integer $\lfloor (n/2) \rfloor$.

5 doub: On input integer n, returns the integer 2*n.

6 log: On input integer n, returns the integer logn if n>0 and returns -1 otherwise.

7 exp: On input integers m, n, returns the integer min(m,$2^n$) if m$\geq$0 and n$\geq$0 and returns 0 otherwise.

8 sub: On input integers m, n, returns the integer m-n.

9 add: On input integers m, n, returns the integer m+n.

10 abs: On input integer n, returns the absolute value of n.

11 sign: On input integer n, returns the integer n/Abs(n) if n$\neq$0 and returns 0 otherwise.

12 if: On input integers l, m, n, returns the integer m if l$\neq$0 and returns n otherwise.

13 nil: This is a function with no argument which returns the empty string.

14 sym: On input integer n, returns the string of length 1 consisting of the $n^{th}$ symbol of the alphabet if n>0 and n≤| Σ | and returns the empty string otherwise.

15 len: On input string x, returns the integer length-of-x.

16 head: On input string x and integer n, returns the string consisting of the first min(len(x),max(n,0)) symbols of x.

17 conc: On input strings x, y, returns the string obtained by concatenating x and y.

18 mask: On input $x\in\Sigma^+$, $y\in\Sigma^*$, mask outputs the (unique) string of length |y| in the set {z|z is a prefix of an element of {x}*}; i.e. it outputs the symbols of x in left-to-right order over and over again till the number of symbols output is |y|.

19 comb: On input $x,y,z\in\Sigma^*$, |x|=|y|=|z|, comb outputs a string of length |x| using the bits of x to select between the bits of y and z. The $k^{th}$ bit output by comb is the $k^{th}$ bit of y if the $k^{th}$ bit of x is 0, and the $k^{th}$ bit of z otherwise.

20 find: On input strings x, y, returns string z of length equal to that of y such that the $n^{th}$ symbol of z is 1 if their is an occurrence of the first $\min(len(x),\lceil(log_2(len(y)+1))\rceil)$ symbols of x as a substring of y starting from the $n^{th}$ symbol of y and is 0 otherwise.

These string manipulation functions are illustrative and other (easy) string operations assumed to be available may be identified from the implementations presented below.

## 4.2 Two Lemmas about Mode Transitions

In presenting the implementation of the basic redistributive functions on the DTM, it will be convenient to use a resource-bounded composition schema which, given efficient implementations of two functions, returns an efficient implementation of their composition. While such schemas are well known for complexity measures like runtime, workspace and reversals, it is not obvious that such a schema exists for mode transitions. Here we prove two lemmas that show the flexibility of this resource for efficient programming.

**Lemma 4.2.1:**

*Let M be any DTM with cycle-time w and let k be any integer greater than 0. Let M' be the*

*DTM obtained from M by padding the tape alphabet of M with enough new symbols so that the cycle-time of M' becomes k*w, with no change in the delta function. Then the computation of M' is the same as that of M on each input, the runtime, workspace and reversal complexities are the same for M and M' and if the mode transition complexity of M on inputs of length n is x(n), then the mode transition complexity of M' is O(x(n)).*

**Proof:**

The only case needing detailed treatment is that of mode transitions. To see that the claim holds, note that the expected head movement remains the same if the mode tuple is lengthened by concatenating several copies of the original mode. Since the computations are identical on each input, one may refer to corresponding steps of the two machines. Let a step at which a mode transition by M occurs be referred to as a cutpoint. Let the steps between cutpoints be referred to as belonging to the immediately previous cutpoint. (Cutpoint steps belong to themselves.) Let the mode transitions of M' occuring at steps belonging to a cutpoint be referred to as belonging to the mode transition of M at that cutpoint. The number of mode transitions of M' belonging to any mode transition of M is clearly at most k*w. The mode transitions of M' accounted for in this way are the only mode transitions made by M'. Thus if the number of mode transitions used by M on inputs of length n is at most x(n), the number of mode transitions used by M' is at most k*w*x(n).

Note that the mode transition complexity of M' can easily be less than that of M. Suppose that M is actually following a cyclic pattern of length k and k is mutually prime to w. Then the mode transition complexity of M could be high while that of M' is low.

∎

**Lemma 4.2.2:**

*Let $M_1$ and $M_2$ compute functions f() and g() and let their mode transition complexities on inputs of length n be $x_1(n)$ and $x_2(n)$ respectively. Then there exists a DTM $M_3$ computing function f(g()) with the number of mode transitions used on input string y being $O(x_1(|g(y)|)+x_2(|y|))$.*

**Proof:**

Construct $M_3$ in the ordinary way and use the construction of the previous lemma to adjust its cycle-time to some common multiple of the cycle-times of $M_1$ and $M_2$.

∎

## 4.3 Implementation of Basic Redistributive Functions on the DTM

The implementation of the basic redistributive functions is presented in pseudocode. The pseudocode convention is as follows:

The language is pseudo-C. Each tape is represented as a one-dimensional array. Each location of the array corresponds to a location of the tape. The numbering of the locations is as per the convention that the head positions at the start of the computation are numbered 0 and incrementing is to the right. (Thus a negative array index represents a head position to the left of the starting position.) Upper case variables represent either constants or variables bound outside the scope of the algorithm being presented. Lower case variables represent machine state and iterator variables bound within the algorithm being presented. Such iterator variables usually refer to head positions of tapes. Note that there are implicit tape rewinds when these iterator variables are initialised.

By definition, sweeps take O(1) mode transitions. To represent these in pseudocode, we introduce the *sweep* pseudo-statement, whose syntax and semantics is the same as the *for* statement, except for the use of the keyword *sweep* in place of the keyword *for*. The use of the sweep pseudo-statement constitutes an assertion that the code embedded in its scope does not cause a mode transition and that the entire for-loop can be implemented with O(1) mode transitions. This distinguishes sweeps from those for-loops in the pseudocode where the embedded code requires one or more mode transitions. In the embedded code of a sweep statement, head positions on different tapes are specified as linear functions of the iterator variable. This indicates how synchronised cycles of head movements on the tapes involved can be set up, reducing the number of mode transitions.

The implementation of the basic redistributive functions is now presented as programs in pseudocode. Each input and the output including each additional input is on a separate tape. The efficient composition schema of the previous section is used implicitly. The use of arithmetic expressions in the pseudocode is to be understood as representing occurrences of the housekeeping functions for arithmetic computation. The portion of tape accessed by a sweep statement may be thought of as a string in which case many of the sweep statements in the pseudocode could be replaced by the string manipulation functions. This is not done here for uniformity.

1 Outer shuffle $so_m$: Using a temporary tape $Array_{new}$, an inplace outer shuffle of the tape $Array_{old}$ is obtained.

begin

sweep(q=0;q<($2^m$/2);q++)$Array_{new}$[q] ←$Array_{old}$[2q];

sweep(q=0;q<($2^m$/2);q++)$Array_{new}$[($2^m$/2)+q]←$Array_{old}$[2q+1];

sweep(q=0;q< $2^m$;q++)$Array_{old}$[q] ←$Array_{new}$[q];

end

2 Inner shuffle $si_m$: Using a temporary tape $Array_{new}$, an inplace inner shuffle of the tape $Array_{old}$ is obtained.

begin

sweep(q=0;q<($2^m$/4);q++)$Array_{new}$[4q] ←$Array_{old}$[4q];

sweep(q=0;q<($2^m$/4);q++)$Array_{new}$[4q+1]←$Array_{old}$[4q+2];

sweep(q=0;q<($2^m$/4);q++)$Array_{new}$[4q+2]←$Array_{old}$[4q+1];

sweep(q=0;q<($2^m$/4);q++)$Array_{new}$[4q+3]←$Array_{old}$[4q+3];

sweep(q=0;q< $2^m$;q++)$Array_{old}$[q] ←$Array_{new}$[q];

end

3 Skew $sk_{c,m}$, for each $2 \leq c \leq m$: The last c bits of the address partition the data into $2^c$ interleaved blocks each of which is a regularly spaced sequence of length $2^{m-c}$ and spacing $2^c$. In a skew $sk_{c,m}$, the last c bits of the address are replaced by the last bit extended c times. The two values of this last bit correspond to two of these $2^c$ blocks, one of which consists of

the first bit of data and every $(2^c)^{th}$ bit thereafter, while the other consists of the $(2^c\text{-}1)^{th}$ bit of data and every $(2^c)^{th}$ bit thereafter. The skew is achieved by interleaving these 2 blocks to form a sequence of $2^{m-c}$ bit-pairs and duplicating each bit-pair $2^{c-1}$ times. Using temporary tapes A, B and $\text{Array}_{new}$, an inplace skew $\text{sk}_{c,m}$ of the tape $\text{Array}_{old}$ is obtained:

```
begin
sweep(q=0;q<2^c;q++)B[q]←0;B[2^c-1]←1;
for(i=0;i<(m-c);i++){
sweep(q=0;q<2^(c+i);q++)A[q]←B[q];
sweep(q=0;q<2^(c+i);q++)A[q+2^(c+i)]←B[q];
sweep(q=0;q<2^(c+i+1);q++)A[q]←B[q];
}
mode=WAITING;
sweep(q=0;q<2^m;q++)switch(mode){
case WAITING:data=Array_old[q];Array_new[q]=data;mode=SKIPPING;break;
case SKIPPING:if(B[q]==1)mode=WAITING; else mode=COPYING;break;
case COPYING:Array_new[q]=data;mode=SKIPPING;break;
}
sweep(q=0;q<2^c;q++)B[q]←0;B[0]←1;
for(i=0;i<(m-c);i++){
sweep(q=0;q<2^(c+i);q++)A[q]←B[q];
sweep(q=0;q<2^(c+i);q++)A[q+2^(c+i)]←B[q];
sweep(q=0;q<2^(c+i+1);q++)A[q]←B[q];
}
mode=WAITING;
sweep(q=(2^m-1);q>=0;q--)switch(mode){
case WAITING:data=Array_old[q];Array_new[q]=data;mode=SKIPPING;break;
case SKIPPING:if(B[q]==1)mode=WAITING; else mode=COPYING;break;
case COPYING:Array_new[q]=data;mode=SKIPPING;break;
```

}

sweep(q=0;q<$2^m$;q++)$Array_{old}$[q]←$Array_{new}$[q];

end

Note that the entire switch statement is first-order involving no head movement and therefore can be executed in one step.

4 Select $selc_m$, for each c∈Σ, i.e. $sel0_m$ and $sel1_m$: The last bit of the address is replaced by c. Using temporary tape $Array_{new}$, an inplace select $selc_m$ of the tape $Array_{old}$ is obtained:

begin

mode=WAITING;

sweep(p=0;p<$2^m$;p++)switch(mode){if(c==0)q=p;else q=($2^m$-1-p);

case WAITING:data=$Array_{old}$[q];$Array_{new}$[q]=data;mode=COPYING;break;

case COPYING:$Array_{new}$[q]=data;mode=WAITING;break;

}

sweep(q=0;q<$2^m$;q++)$Array_{old}$[q]←$Array_{new}$[q];

end

5 Successor $suc_{c,m}$, for each 2≤c≤m: Increment if there is no carry out of the $c^{th}$ bit (i.e. saturating increment of the last c bits). Using a temporary tape $Array_{new}$, an inplace successor of the tape $Array_{old}$ is obtained.

begin

for(i=0;i<(m-c);i++)$so_m$;

/* This separates the interleaved blocks */

sweep(q=0;q<$2^m$;q++)$Array_{new}$[q]=$Array_{old}$[q+$2^c$];

/* This does the job for all but the last value of the c-tuple */

/* Since SUC($2^c$)=$2^c$ itself, junk data got written in the last block */

/* Recover from this by restoring the old data in the last block */

sweep(q=0;q<$2^c$;q++)$Array_{new}$[$2^m$-1-q]=$Array_{old}$[$2^m$-1-q];

sweep(q=0;q<$2^m$;q++)$Array_{old}$[q]=$Array_{new}$[q];

/* Now the separated blocks have to be interleaved again */

for(i=0;i<c;i++)$so_m$;

end

6 Double $db_{c,m}$, for each $2 \leq c \leq m$: Saturating double of the last c bits. Using temporary tapes $Array_{new}$, $Array_1$ and $Array_2$, an inplace double of the tape $Array_{old}$ is obtained:

begin

$(so_m)^{m-c+1}(so_m si_m)^{c-1}$;

/* This brings data from even addresses to the front in each block */

/* The last $2^{c-1}$ elements in each block of size $2^c$ are wanted */

$(so_m)^{m-c}$; /* This separates the interleaved blocks */

/* The second half of $Array_{old}$ becomes the first half of $Array_{new}$ */

sweep(q=0;q<$2^{m-1}$;q++)$Array_{new}$[q]=$Array_{old}$[q+$2^{m-1}$];

/* Since DOUBLE(x)=$2^c$ for all x in the range $2^{c-1} \leq x \leq 2^c$, */

/* The last data element of the original block has to be copied here */

/* this is done simultaneously for all blocks as follows: */

/* The last valid data block starts at offset ($2^c$-1)*$2^{m-c}$ */

/* Copy this for later use and make $2^{c-1}$ copies: */

j=($2^c$-1);

for(i=0;i<(m-c);i++)j=2*j;

sweep(q=0;q<$2^{m-c}$;q++)$Array_1$[q]=$Array_{old}$[q+j];

l=$2^{m-c}$;

for(j=0;j<(c-1);j++){

sweep(q=0;q<l;q++)$Array_2$[l+q]=$Array_1$[q];l=2*l;

sweep(q=0;q<l;q++)$Array_1$[q]=$Array_2$[q];

}

/* The contents of $Array_1$ become the second half of $Array_{new}$ */

sweep(q=0;q<$2^{m-1}$;q++)$Array_{new}$[q+$2^{m-1}$]=$Array_1$[q];

sweep(q=0;q<$2^m$;q++)$Array_{old}$[q]=$Array_{new}$[q];

$(so_m)^c$; /* The separated blocks are interleaved again */

/* The output is obtained in $Array_{old}$ itself */

end

7 Log Truncate $trlog_{c,m}$, for each $2 \leq c \leq m$. Using a temporary tape $Array_1$, an inplace double of the tape $Array_{old}$ is obtained:

begin

$(so_m)^{m-c}$; /* This separates the interleaved blocks */

/* The first (c-1) blocks each of size $2^{m-c}$ are now correct */

/* The last valid data block now starts at offset $(c-1)*2^{m-c}$ */

/* Copy this for later use and make $2^c$ copies: */

j=(c-1);

for(i=0;i<(m-c);i++)j=2*j;

sweep(q=0;q<$2^{m-c}$;q++)$Array_1$[q]=$Array_{old}$[q+j];

l=$2^{m-c}$;

for(j=0;j<c;j++){

sweep(q=0;q<l;q++)$Array_2$[l+q]=$Array_1$[q];l=2*l;

sweep(q=0;q<l;q++)$Array_1$[q]=$Array_2$[q];

}

j=c;

for(i=0;i<(m-c);i++)j=2*j;

sweep(q=0;q<$2^m$;q++)$Array_{old}$[q+j]=$Array_1$[q];

$(so_m)^c$; /* The separated blocks are interleaved again */

/* The output is obtained in $Array_{old}$ itself */

end

8 Const Truncate $tr2_{c,m}$, for each $2 \leq c \leq m$: The implementation of $tr2_{c,m}$ is similar to that of $trlog_{c,m}$.

The implementation of the housekeeping functions is straightforward. Details are omitted. The claims of efficiency follow readily from the implementations.

## 4.4 Relational Transformations

Consider the calculus FO+Y(T,S,V) with T=LOG, S=LIN and V∈{LOG,BIN}. Let the binding of a k-ary predicate P over S be represented on a DTM worktape as follows: The truth-values of the predicate for all tuples s=$< s_1 s_2 \cdots s_k >$ are laid out as YES/NO bits (YES=1;NO=0) on this tape from left to right in lexicographic order of s, with $s_1$ being the most significant variable over S and $s_k$ being the least significant variable over S for this purpose. For example, if P($S1^{k-1}$*SUC(S1)) is true, then the second non-blank symbol from the left is 1. When S∞ is not a power of 2, gaps are left in the data structure for technical reasons. This is done as follows:

Let Z be a totally ordered domain of size $2^{LOG\infty}$ with first element Z1 and last element Z∞. In general, Z∞≥LIN∞=S∞. Let LIN2Z(.) be the function embedding LIN as the prefix of Z. The truth-values P(s) are laid out at locations indexed by LIN2Z(s), using a total of $Z\infty^k$ rather than $S\infty^k$ tape cells. Note, however, that $Z\infty^k$=O($S\infty^k$).

With this representation, the position of the head on the tape constitutes an encoding of a particular tuple s (ignoring the gaps). The truth-value of the predicate on a tuple s may be obtained by positioning the tapehead at the corresponding location and reading the contents of the tape cell under the head.

Consider a k-tuple x of constants, variables and derived terms of domain S. In general, x contains occurrences of constants and variables over the BIN, LOG and LIN domains, with the proviso that the only variables over the LIN domain that occur in x are those of s. The tuple x gets bound to specific ground tuples when the variables of s are given values (in the context of given bindings for the other variables in x, referred to as a suitable context). The lexicographically ordered sequence of all the tuples s induces an ordered sequence of ground tuples x in a suitable context (possibly with multiple occurrences of some ground tuples in the induced sequence). The truth values of the predicate for this ordered sequence of ground tuples x can be laid out as YES/NO bits on a DTM worktape (with gaps if necessary) in the manner described above.

Such a tuple **x** is said to be a *relational transformation.*

The $Z\infty^k$-bit string representing the binding of a k-ary predicate P is the input. As **s** varies lexicographically from $S1^k$ to $S\infty^k$, **x** addresses various locations in the input. The data at these locations (in the order of access) has to be written on the output tape to achieve the transformation. This constitutes a transformation of $S\infty^k$-bit input sequences into $S\infty^k$-bit output sequences (ignoring the ($Z\infty^k$-$S\infty^k$) bits in the gaps). Each such k-tuple **x** specifies a relational transformation and the set of all such tuples specifies the class of relational transformations. One may say that the input sequence is in s-order and the task is to generate an output sequence in x-order.

Since **x** is a k-tuple of elements drawn from the Herbrand Universe of constants, the (bound) variables of the BIN and LOG domains (given as a suitable context), the (unbound) variables of s and predefined functions (which are unary), one can obtain a general transformation (of this class) by function composition from two types of simpler transformations, as shown below.

Let **s'** be a k-tuple obtained by permuting the elements of **s**. The transformation specified by **s'** is a permutation on $S\infty^k$-bit sequences, since the pair <s,s'> defines a permutation over numbers in the range [1-$S\infty^k$].

Let **s'** be a k-tuple obtained by replacing the last c (for some $1 \leq c \leq k$) elements of **s** by a c-tuple of terms drawn from the Herbrand Universe of constants, the (bound) variables of **t** and **r**, the (unbound) variable $s_k$ and the predefined functions (which are unary). ($s_k$ is the $k^{th}$ element of **s**). Then the pair <s,s'> defines a function b(.) over numbers in the range [1-$S\infty^k$] having the property that $m \leq n$ iff $b(m) \leq b(n)$. The transformation of $S\infty^k$-bit sequences specified by **s'** is said to be a c-broadcast.

Permutations are lossless while broadcasts are order-preserving. Any particular **x** can be decomposed into permutations and broadcasts. Broadcasts require the identification of a single sequence of length $S\infty^{(k-c+1)}$ and duplication of each bit $S\infty^{(c-1)}$ times.

Permutations are shown to be further decomposable into relational outer shuffles and relational inner shuffles. The relational outer shuffles and relational inner shuffles referred

to here are variants of the outer and inner shuffles defined among the basic redistributive functions for data manipulation. There, addresses are binary, and a string of $2^m$ bits uses m-bit addresses. Now consider a variant in which the address is k digits and each digit is S$\infty$-ary, giving an array-size of $S\infty^k$.

Define the relational outer shuffle $rso_k$ as:

$rso_k$: $Array_{new}[s_1s_2\cdots s_{k-2}s_{k-1}s_k] \leftarrow Array_{old}[s_2\cdots s_{k-2}s_{k-1}s_ks_1]$

and the relational inner shuffle $rsi_k$ as:

$rsi_k$: $Array_{new}[s_1s_2\cdots s_{k-2}s_{k-1}s_k] \leftarrow Array_{old}[s_1s_2\cdots s_{k-2}s_ks_{k-1}]$

to be used as primitive transformations. It is easy to see that the permutations of interest can be obtained by function composition from $rso_k$ and $rsi_k$.

When k=1, there are no non-trivial permutations and the only transformations are broadcasts with c=1. When k=2, the outer shuffle and inner shuffle permutations are identical:

$rso_2$=$rsi_2$: $Array_{new}[s_1s_2] \leftarrow Array_{old}[s_2s_1]$

With the standard representation of a matrix as a vector, this corresponds to:

For $1 \le i,j \le S\infty$, $rso_2$=$rsi_2$: $Array_{new}[S\infty*(i-1)+j] \leftarrow Array_{old}[S\infty*(j-1)+i]$.

It is not obvious how this can be computed within the given resource bounds. However, gaps have been left in the data structure, so it is actually enough to compute:

For $1 \le i,j \le Z\infty$, $rso_2$=$rsi_2$: $Array_{new}[Z\infty*(i-1)+j] \leftarrow Array_{old}[Z\infty*(j-1)+i]$.

This is achieved by the following algorithm:($Z\infty=2^{LOG\infty}$)

**for(p=1;p$\le$LOG$\infty$;p++){$so_{2*LOG\infty}$(Array$_{old}$,Array$_{new}$);}**

To see why this works, note that each element of the tuple s can be written as a LOG$\infty$-tuple of binary variables, so that s is equivalent to a (k*LOG$\infty$)-tuple of binary variables. Such a tuple can be treated as a "node address" for an interconnection network, with m=k*LOG$\infty$, which allows us to use the two primitives, $so_m$ and $si_m$. Now the transformation $rso_2$ can be seen to be simply $so_{2*LOG\infty}{}^{LOG\infty}$, i.e. $so_{2*LOG\infty}$ iterated LOG$\infty$ times. In the algorithm above, each of the LOG$\infty$ iterations computes $so_{2*LOG\infty}$. This generalises to k>2, so that the following algorithm computes $rso_k$ for k>1:

$rso_k = so_{k*LOG\infty}{}^{LOG\infty}$.

To compute $rsi_k$, let $so^{-}_{k*LOG\infty}$ denote the inverse of $so_{k*LOG\infty}$:

inverse($so_{k*LOG\infty}$)=$so_{k*LOG\infty}{}^{(k-1)*LOG\infty}$. (More efficient implementation is possible.) Then

$rsi_k=so_{k*LOG\infty}{}^{LOG\infty}[so^{-}_{k*LOG\infty}(si_{k*LOG\infty}so_{k*LOG\infty})^{LOG\infty-1}si_{k*LOG\infty}(so^{-}_{k*LOG\infty})^{LOG\infty-1}]^{LOG\infty}$

This is in postfix notation, i.e. first $so_{k*LOG\infty}$ is to be iterated LOG$\infty$ times, then $so^{-}_{k*LOG\infty}$ is to be computed once, and so on.

$rso_2=rsi_2$ is just a matrix transposition: the input is in row-major form and the output is required in column-major form. For k>2 also, one may think of $rso_k$ and $rsi_k$ as transpositions of multi-dimensional arrays.

This covers the permutations. Next it is shown how to compute broadcasts. Two cases are separately considered: those c-broadcasts in which c=1 and those c-broadcasts in which each of the last c elements of s′ is just $s_k$. The first type of broadcast is referred to as a calculation and the second type of broadcast is referred to as a relational skew. It can be seen that an arbitrary broadcast can be decomposed into a functional composition of permutations, calculations and relational skews. Further, in view of the gaps in the data structure, it can be seen that relational skew can in turn be decomposed into a functional composition of permutations and primitive skews.

In a calculation, the last element of s is replaced by an essentially constant term or a function of $s_k$ to give s′. (An essentially constant term has a constant or a (bound) variable over the BIN or the LOG domain where the binding is given as part of a suitable context. The predefined functions may also occur.) If the last element of s′ is constant or essentially constant, in view of the gaps in the data structure, it can be decomposed into a functional composition of permutations and selects.

If the last element of s′ is a function of $s_k$ (i.e. an element of the Herbrand Universe of the (unbound) variable $s_k$ and the (unary) predefined functions), then as $s_k$ varies from S1 to S$\infty$, the last element of s′ selects regularly spaced subsequences of size S$\infty^{(k-1)}$ for each value of $s_k$. The calculation is achieved by interleaving these S$\infty$ subsequences to form a sequence of length S$\infty^k$. In view of the gaps in the data structure, the algorithm actually

interleaves Z∞ subsequences of length $Z\infty^{k-1}$ each. There is a set of primitive algorithms in terms of which each predefined function (TOBIN, FROMBIN, TOLOG, FROMLOG, SUC and DOUBLE) and their functional compositions can be expressed.

The algorithm for the function SUC over the domain LIN is obtained in terms of the primitive algorithm for $suc_{c,m}$ and the masking and combine functions.

The algorithm for the function DOUBLE over the domain LIN is obtained in terms of the primitive algorithm for $db_{c,m}$ and the masking and combine functions.

The algorithm for the function FTLOG(.)=FROMLOG(TOLOG(.)) is obtained in terms of the primitive algorithm for $trlog_{c,m}$ and the masking and combine functions.

The algorithm for the function FTBIN(.)=FROMBIN(TOBIN(.)) is obtained in terms of the primitive algorithm for $tr2_{c,m}$ and the masking and combine functions.

## 4.5 Evaluation of FO+Y(T,S,V) Formulae on the DTM

**Theorem 4.5.1:**

*Let F be a formula with time domain T, space domain S, variance domain V and space-arity k≥1 (such that T and S are not both LOG). Then for a constant m depending on F but independent of the input size n, there is a DTM M operating in $O(V\infty^m S\infty^k)$ workspace and $O(T\infty^m)$ mode transitions which recognizes the language expressed by F.*

**Proof:**

When the time domain is LIN, a naive simulation satisfies the constraints; since S∞≥LOG∞ and k≥1, the workspace used for indexes, pointers, counters etc. is $O(S\infty^k)$. The only case needing detailed treatment is T=LOG,S=LIN,V∈{LOG,BIN}. W.l.o.g. it is assumed that the given formula is in standard form (Y(LOG,LIN,V)[P,**t**,**r**,**s**](F)[**v**] with time arity c, variance arity q and space arity k).

For constructing the predicate defined by the Y-operator there is a "main" worktape. The current truth-values of the predicate P for all tuples **r**∗**s** are laid out as YES/NO bits (YES=1;NO=0) on this tape from left to right in lexicographic order of **r**∗**s**, with $r_1$ being

most significant and $s_k$ being least significant. When $S\infty$ is not a power of 2, gaps are left in the data structure. This is done n the same manner as described in the previous section. The truth-values P(r*s) are laid out at locations indexed by r*LIN2Z(s), using a total of $V\infty^q Z\infty^k$ rather than $V\infty^q S\infty^k$ tape cells. Note, however, that $V\infty^q Z\infty^k = O(V\infty^q S\infty^k)$.

Corresponding to each occurrence P(x) of the predicate in the formula F, there is an auxiliary worktape. In general, x contains occurrences of the variables of t, r and s and gets bound to specific ground tuples when t, r and s are given values. Let t be a tuple of constants. The lexicographically ordered sequence of all the tuples r*s induces a sequence of ground tuples x (possibly with multiple occurrences of some ground tuples). The truth values of the predicate for this particular occurrence with this ordered sequence of ground tuples x are laid out as YES/NO bits on this auxiliary tape (with gaps if necessary). Again this is similar to the previous section, with the proviso that here terms over the variance domain also occur in the tuple.

The predefined predicates INP and EQ are handled in a similar manner. For each occurrence of these predicates with argument tuple x, an auxiliary worktape is used on which the truth values of the predicate for the sequence of ground tuples x induced by the lexicographically ordered sequence of all tuples r*s for a given tuple of constants t are stored. $O(V\infty^q S\infty^k)$ space is used on the main worktape and on each of the auxiliary worktapes for this storage.

The computation consists of an initialisation phase, a construction phase of $T\infty^c$ iterations and a reporting phase. In the initialisation phase, the main worktape is written with NO bits corresponding to an empty relation (everywhere false predicate). Each iteration of the construction phase consists of an indexing stage and a commit stage. In each indexing stage the contents of the auxiliary worktapes are updated to match the current contents of the main worktape. (The iteration number fixes the value of t).

In each commit stage, the heads of the main worktape and the auxiliary worktapes make a single simultaneous sequential (left to right) sweep without any mode transitions. The position of the head corresponds to the instantaneous value of the tuple r*s. Since the

formula is in standard form and the truth-values of the predicate occurrences are available at the auxiliary head positions, the new truth-value of the predicate P for tuple **r*s** can be computed in O(1) time, i.e. by the delta function of the DTM itself. This new value is written on the main worktape at the current location of the head.

Finally, in the reporting phase, the final value of the predicate P constructed on the main worktape is tested on the tuple **v**, and the DTM accepts iff this test succeeds. In pseudo-code, this algorithm may be summarised as follows:(M1, M2 and M3 are the number of occurrences of P, INP and EQ respectively and pf is the propositional formula in F))

**function**(INP:**predicate-over**(S)):**bool**

**tape-alloc** P,PAux1,$\cdots$,PAuxM1,

InpAux1,$\cdots$,INPAuxM2,

EqAux1,$\cdots$,EqAuxM3:**predicate-over**($V^qS^k$);

**begin** P$\leftarrow${};

**for** t=$T1^c$ **to** $T\infty^c$ **do**

**begin** PAux1$\leftarrow$update-PAux1(P);$\cdots$PAuxM1$\leftarrow$update-PAuxM1(P);

InpAux1$\leftarrow$update-InpAux1(INP);$\cdots$InpAuxM2$\leftarrow$update-InpAuxM2(INP);

EqAux1$\leftarrow$update-EqAux1();$\cdots$EqAuxM3$\leftarrow$update-EqAuxM3();

P$\leftarrow$pf(PAux1,$\cdots$,PAuxM1,InpAux1,$\cdots$InpAuxM2,EqAux1,$\cdots$,EqAuxM3);

**end**;

**if**(P(**v**))return(TRUE); else return(FALSE);

**end.**

To complete the proof it remains to show how the update of the auxiliary worktape contents in the indexing stage is achieved. This has to be done within the resource bounds claimed in the statement of the theorem.

First the update of the auxiliary tapes for the predicate P is described. Let **x** be a (q+k)-tuple that is part of an occurrence of P in F. As per assumption, the first q components of **x** do not contain any occurrence of the variables of **s**. It is shown that the update of the auxiliary worktape corresponding to this occurrence can be achieved in simultaneous $O(V\infty^qS\infty^k)$

space and poly(T$\infty$) mode transitions. The value of t is fixed during an update.

Let the last k components of x constitute the k-tuple x'. The update algorithm consists of V$\infty^q$ iterations, indexed by r. Each iteration has the values of t and r fixed, so the first q components of x are essentially constant terms. The contiguous Z$\infty^k$-bit section of the main worktape indexed by these q components is identified. As s varies lexicographically from S$1^k$ to S$\infty^k$, x' addresses various locations in this section (during one iteration). The data at these locations (in the order of access) have to be written on the auxiliary tape to complete one iteration of the update algorithm. This constitutes a relational transformation in the sense of the previous section and may be implemented as such.

This completes the description of the update of the auxiliary tapes for the predicate P. The update of the auxiliary tapes for the predicate INP is handled in exactly the same manner. The update of the auxiliary tapes for the predicate EQ is also handled in exactly the same manner when k>1. It remains to show how the update of the auxiliary tapes for the predicate EQ is handled for the case k=1 to complete the proof of the theorem.

EQ is a binary predicate. Let k=1. Both the arguments of EQ are terms drawn from the Herbrand Universe of constants, essentially constant terms, the (unbound) variable $s_1$ and the predefined functions. When both arguments are constant or essentially constant, the update is carried out by writing Z$\infty$ copies of the truth-value of their comparison on the auxiliary tape.

When one of the arguments is a constant or essentially constant term E and the other is a term containing $s_1$, the update is carried out in three steps. First the auxiliary tape is written with Z$\infty$ copies of FALSE(=0). Next the value TRUE(=1) is written at location $s_1$=E. Finally, a calculation specified by the term containing $s_1$ is carried out to move this truth-value to its desired position.

When k=1, and both arguments of EQ are terms containing $s_1$, there is only nearly linear workspace, so the technique of separately computing the broadcasts for each variable of s and then computing a skew cannot be used. Now use the fact that the (unary) functions SUC, DOUBLE, FTLOG and FTBIN are non-decreasing. As noted in the previous section,

the terms containing $s_1$ can be written using these functions alone. Since the domain LIN is finite, for each term $E(s_1)$ there is a smallest element $l \in LIN$ such that for all $s_1 \geq l$, $E(s_1)=E(l)$.

First this element is identified for each of the two terms containing $s_1$. Let the two elements so obtained be $l_1$ and $l_2$. W.l.o.g. assume $l_1 \leq l_2$. For $l_2 \leq s_1 \leq LIN\infty$, the approach of comparing two essentially constant terms is used. For $l_1 \leq s_1 < l_2$, the approach of comparing a term containing $s_1$ with an essentially constant term is used. It remains to describe the approach when $LIN1 \leq s_1 < l_1$.

In this range, all occurrences of FTLOG and FTBIN in the terms can be replaced by the identity function and the function DOUBLE does not saturate. The resulting simplified terms contain only SUC and DOUBLE and the occurrence $EQ[E_1(s_1),E_2(s_1)]$ describes a linear equation with one unknown $s_1$ which is restricted to be a natural number less than $l_1$. It is clear that either this requirement is unsatisfiable (for example, $EQ[s_1,SUC(s_1)]$ is false everywhere in the range $LIN1 \leq s_1 < l_1$), or valid (for example, $EQ[SUC(SUC(DOUBLE(s_1))),DOUBLE(SUC(s_1))]$ is true everywhere in the range $LIN1 \leq s_1 < l_1$) or has a unique solution $l \geq 1$ in the natural numbers which depends only on the term and not the input. The update is easy after comparing this precomputed value l with the input-dependent number $l_1$.

This completes the description of the algorithms for updating the auxiliary worktape contents in the indexing stage of the construction phase of the computation of DTM M on input x. It can be seen that M operates in $O(V\infty^m S\infty^k)$ workspace and $O(T\infty^m)$ mode transitions for some m independent of the input size n and recognizes the language expressed by formula F, thus proving the theorem.

∎

# Chapter 5

# The main result

In chapter 2, the sequential and parallel models of computation and the resources of interest were defined. Then, the resource bounded simulation of the DTM model on the PRAM model was presented. In chapter 3, the calculus FO+Y(T,S,V) extending first order logic was defined and a standard form was demonstrated. Then, the simulation of the PRAM model in the calculus was presented. The evaluation of formulas in standard form on the DTM model was presented in chapter 4. In this chapter, first, these results are combined to give an equivalence between the DTM (with mode transitions) and the PRAM for *conventional resource bounds* (modulo polynomial factors). Then this equivalence is extended to other resource bounds. This equivalence holds for decision problems. The next section extends the equivalence to function problems. This shows that the class NC can be characterised in terms of mode transitions and workspace on the DTM. A unified invariance thesis and a conjecture are stated. Finally, two corollaries relating this thesis and conjecture to the first and second machine classes are proved.

## 5.1 Equivalence modulo polynomial factors for Conventional Resource Bounds

In chapter 2, conventional resource bounds for the DTM were defined as follows:

Conventional workspace bounds s(n) and conventional mode transition bounds x(n) are those of the form $O(\log(n)^k)$, $O(n^k)$ and $O^*(n^k)$ (for some constant $k \geq 1$), with the restriction that either $x(n)=\Omega(n)$ and $s(n)=O(\text{poly}(x(n)))$ or $s(n)=\Omega(n)$ and $\log(s(n))=O(\text{poly}(x(n)+\log(n)))$.

The following two results were proved:

**Theorem 5.1.1:**

*For conventional workspace bound s() and conventional mode transition bound x(), such that $x(n)=\Omega(n)$ and $s(n)=O(\text{poly}(x(n)))$, let M be a DTM that uses at most s(n) workspace and x(n) mode transitions on any input of length n and accepts language L. Then there is a PRAM algorithm that uses at most $h(n)=O(s(n)+\log(n))$ hardware and $t(n)=O(\text{poly}(x(n)+\log(n)))$ parallel time and accepts L.*

**Theorem 5.1.2:**

*For conventional workspace bound s() and conventional mode transition bound x(), such that $s(n)=\Omega(n)$ and $\log(s(n))=O(\text{poly}(x(n)+\log(n)))$, let M be a DTM that uses at most s(n) workspace and x(n) mode transitions on any input of length n and accepts language L. Then there is a PRAM algorithm that uses at most $h(n)=O(s(n)+\log(n))$ hardware and $t(n)=O(\text{poly}(x(n)+\log(n)))$ parallel time and accepts L.*

In chapter 3, conventional resource bounds for the PRAM were defined as follows:

Conventional hardware bounds h(n) and conventional parallel time bounds t(n) are those of the form $O(\log(n)^k)$, $O(n^k)$ and $O^*(n^k)$ (for some constant $k \geq 1$), with the restriction that either $t(n)=\Omega(n)$ and $h(n)=O(\text{poly}(t(n)))$ or $h(n)=\Omega(n)$ and $\log(h(n))=O(\text{poly}(t(n)+\log(n)))$.

The following two results were proved:

**Theorem 5.1.3:**

*For $T,S \in \{LIN, LOG\}$ (such that T and S are not both LOG) and $V \in \{LOG, BIN: V\infty < S\infty \wedge V\infty \leq T\infty\}$, let A be a CRCW PRAM decision algorithm written using only READ, WRITE, ADD, SUBTRACT, MOVE, INC/DEC, BLT and HALT instructions, which operates with $O(\text{poly}(V\infty * S\infty))$ hardware*

*and poly(T∞) parallel time. Then, there exists a formula of FO+Y(T,S,V) which expresses the language decided by A.*

**Theorem 5.1.4:**

*Given any formula of the calculus FO+Y(T,S,V) (such that T and S are not both LOG), one can construct an equivalent formula in standard form.*

In chapter 4, the following result was proved:

**Theorem 5.1.5:**

*Let F be a formula with time domain T, space domain S, variance domain V and space-arity k≥1 (such that T and S are not both LOG). Then, for some constant m depending on F but independent of the input size n, there is a DTM M operating in $O(V\infty^m S\infty^k)$ workspace and $O(T\infty^m)$ mode transitions which recognizes the language expressed by F.*

Putting these results together, we obtain the following theorem:

**Theorem 5.1.6:**

*For T,S∈{LIN,LOG} (such that T and S are not both LOG) and V∈{LOG,BIN} (such that V∞<S∞ and V∞≤T∞), the calculus FO+Y(T,S,V), the DTM restricted to O(poly(V∞*S∞)) workspace and poly(T∞) mode transitions, and the PRAM restricted to O(poly(V∞*S∞)) hardware and poly(T∞) parallel time, all recognise exactly the same class of languages.*

## 5.2 Equivalence modulo polynomial factors for Acceptable Resource Bounds

To generalise the result of the previous section to other resource bounds, some restriction on the nature of resource bounds considered is necessary for the proof to go through.

**Definition 5.2.1:**

A *simultaneous complexity model* may be said to be a 3-tuple $<M,T_m,H_m>$, where M is a model of computation and $T_m$ and $H_m$ are two complexity measures on M. An $<M,T_m,H_m>$-computation is said to use at most (t,h) resources (equivalently, work within resource bound (t,h)) if it uses at most t(n) units of the resource $T_m$

**and at most h(n) units of the resource $H_m$ on inputs of length n. A resource bound (t,h) on <M,$T_m$,$H_m$> is said to be constructible on the simultaneous complexity model if there is an <M,$T_m$,$H_m$>-machine which on every input of length n uses at most (O(t),O(h)) resources and computes the string $1^{h(n)}0z$ where z is the binary encoding of t(n). It is said to be acceptable if it is constructible on the simultaneous complexity model and one of the following two conditions holds:**

**(1)t=Ω(n) and h=O(poly(t)),**

**(2)h=Ω(n) and log(h)=O(poly(t+log(n))).**

Let $T_{pram}$ and $H_{pram}$ denote the resources parallel time and hardware respectively on the PRAM and let $X_{dtm}$ and $S_{dtm}$ be the resources mode transitions and workspace respectively on the DTM. Then <PRAM,$T_{pram}$,$H_{pram}$> and <DTM,$X_{dtm}$,$S_{dtm}$> are simultaneous complexity models. It is clear that a pair of conventional resource bounds satisfying the two conditions above are constructible on the simultaneous complexity model and hence acceptable. Let such a pair be called a *conventional pair.*

The results proved so far show that every <DTM,$X_{dtm}$,$S_{dtm}$>-computation taking (t,h) resources for some conventional pair (t,h) can be simulated by a <PRAM,$T_{pram}$,$H_{pram}$>-computation taking (poly(t+log(h)),O(h+log(t))) resources and every <PRAM,$T_{pram}$,$H_{pram}$>-computation taking (t,h) resources for some conventional pair (t,h) can be simulated by a <DTM,$X_{dtm}$,$S_{dtm}$>-computation taking (poly(t+log(h)),poly(h+log(t))) resources.

We would like to extend this from conventional resource bounds to all acceptable resource bounds. This requires the generalisation of each of the three simulations of chapters 2, 3 and 4. These are considered in turn.

The proofs of the theorems of chapter 2 simulating the <DTM,$X_{dtm}$,$S_{dtm}$> on the <PRAM,$T_{pram}$,$H_{pram}$> carry through without any change for acceptable resource bounds.

To extend the result of chapter 3, four new domains HW, PT, LOGHW and LOGPT are added to the structures along with corresponding SUC, DOUBLE and typecasting functions, constants and predicate EQ. The domain HW by definition has a size equal to the hardware bound and the domain PT by definition has a size equal to the parallel time bound. The

domains LOGHW and LOGPT by definition have sizes equal to the logarithm of the sizes of domains HW and PT respectively. This gives us structures with seven domains, namely BIN, LIN, LOG, HW, LOGHW, PT and LOGPT. The Y-operator is constrained to have time domain T=PT, space domain S=HW and variance domain V∈BIN,LOGHW. With these modifications, the normal form results as well as the expression of $<PRAM,T_{pram},H_{pram}>$ computations in the calculus go through.

To simulate this new calculus on the $<DTM,X_{dtm},S_{dtm}>$, the only extra work required is to show that the resource bound is constructible on the simultaneous complexity model. This, however, holds by assumption.

## 5.3 Extension from Decision Problems to Function Problems

To handle functions on the $<DTM,X_{dtm},S_{dtm}>$, one needs an output tape. As usual, the output tape is one-way write-only and the space used on the output tape is not counted. However, the mode transitions made on the output tape due to stops and starts are counted.

On the $<PRAM,T_{pram},H_{pram}>$, one needs a write-only area in global memory. The convention we follow is that negative addresses on global memory write operations refer to this area. Since the convention for calculating number of global memory cells used refers to the largest (most positive) address involved in a global memory operation, the locations used for output are implicitly ignored. The convention for interpreting the contents of negative memory as a string is as follows: The string is stored "backward"; i.e. the contents of location at address -1 are the leftmost. The string is encoded in the binary alphabet {01,11} and terminated by either 00 or 10. The contents of negative memory are assumed to be all 0s initially. This convention ensures that unreasonably long output strings cannot be produced by writing to locations with large negative addresses (since the first location left unaccessed during the computation would then contain 00 or 10, thus terminating the string).

In the calculus, we use free variables to encode the output string. An infinite sequence of variable symbols is defined over each of the domains. The rank of the symbols within

the infinite sequence determines their relative significance, with a fixed order of precedence for variables over different domains. With this convention, a formula with these as the only free variables defines a predicate on each input structure, with a canonical lexicographic ordering over tuples. The truth values of this predicate when laid out as bits according to the lexicographic ordering of the tuples gives a binary string. This binary string is assumed to encode an output string with the binary alphabet {01,11} terminated by either 00, 10 or END-OF-STRING.

The simulation of a <DTM,$X_{dtm}$,$S_{dtm}$> computing a function on the <PRAM,$T_{pram}$,$H_{pram}$> proceeds in the same way as in the case of language recognition. At the time of updating the tape contents, the number of symbols written during the sweep is either 0 or equal to the length of the sweep.

To express <PRAM,$T_{pram}$,$H_{pram}$> computations in the extended calculus, a number of free variables which just suffices to encode the <PRAM,$T_{pram}$,$H_{pram}$> output are used to build the formula. Since the output area is write-only, all the truth values required can be expressed by a single formula.

The simulation of the calculus on the <DTM,$X_{dtm}$,$S_{dtm}$> is straightforward when the number of mode transitions permitted is large enough to enable sequential simulation. Otherwise, the output string is produced in a series of iterations, in each of which segments of output are first obtained on a worktape, then transfered to the output tape. The length of segments is limited by the workspace bound and the same workspace is reused in each iteration. The decoding of the output string can be done in O(1) mode transitions for each segment and the number of iterations is small.

## 5.4 Reducing the Complexity Overhead of Simulation

These results show that when an algorithm A on one model that works within resource bound (h(n),t(n)) is simulated on another model, the simulation S(A) works within resource bound $(O(h(n)+\log(n))^k, O((t(n)+\log(n))^k))$ for some constant k. Here, k appears to depend on A.

However, it is easy to see that it can be made independent of A.

**Theorem 5.4.1:**

*Consider both the simulation of DTMs on PRAMs and the simulation of PRAMs on the DTM. There exists an a priori constant k and a simulation method S such that for any algorithm A that works within resource bound (h(n),t(n)), the simulation S(A) works within resource bound* $(O(h(n)+log(n))^k, O((t(n)+log(n))^k))$.

**Proof:**

Each of the three simulations has to be considered in turn. The simulation of the <DTM,$X_{dtm}$,$S_{dtm}$> on the <PRAM,$T_{pram}$,$H_{pram}$> goes through as presented in chapter 2, with a careful complexity analysis.

In chapter 3, the definition of the Y-operator has to be changed. Hitherto, we have allowed a variance domain for space but none at all for time. Now we require the variance domain for space to be BIN and introduce a variance domain for time, which should also be BIN, so that a location in space is represented by a tuple which is an element of $BIN^{k_1}*HW^{k_2}$ while an instant of time is represented by a tuple which is an element of $BIN^{c_1}*P'T^{c_2}$, where $k_1$, $k_2$, $c_1$ and $c_2$ are constants. The normal forms can be achieved without changing $k_2$, and permitting increase in $c_2$ only to handle nesting.

With these modifications, the increase in $c_2$ is bounded by the nesting depth of Y-operators and first-order quantifiers and the number of occurrences of variables of domain HW in terms of type BIN. The assumption is made that <PRAM,$T_{pram}$,$H_{pram}$> algorithms consist of a single par-for-loop enclosed in a single seq-for-loop. This assumption is justified by recourse to a replication theorem [Blelloch'sBook1990]. With this assumption and the modifications mentioned above, we can show a constant k such that that the formula in standard form expressing a <PRAM,$T_{pram}$,$H_{pram}$> computation has space arity at most k times the exponent of the hardware complexity of the <PRAM,$T_{pram}$,$H_{pram}$> algorithm and time arity only an apriori fixed additive constant more than the exponent of the parallel time complexity of the <PRAM,$T_{pram}$,$H_{pram}$> algorithm. (Immerman has shown a particular degree of blowup (i.e. from c to 2c+2) to be sufficient for the hardware [Immerman1987b].)

The extension of the simulation of chapter 4 to the modified calculus is straightforward. The constant k of the theorem is chosen as the maximum over all three simulations.

∎

## 5.5 A Unified Invariance Thesis and a Conjecture

In [VanEmdeBoasHandbook1990], the classical Invariance Thesis and the Parallel Computation Thesis were presented in the following terms:

*Invariance Thesis*: "Reasonable" machines can simulate each other within a polynomially bounded overhead in time and a constant-factor overhead in space.

*Parallel Computation Thesis*: Whatever can be solved in polynomially bounded space on a reasonable sequential machine can be solved in polynomially bounded time on a reasonable parallel machine and vice versa.

The *orthodox interpretation* (both resource restrictions apply simultaneously) was adopted for the Invariance Thesis. The neutral term *machine class* was proposed to replace the evaluative term *reasonable* and the first and second machine class were defined as follows:

The *first machine class* consists of those sequential models which satisfy the Invariance Thesis with respect to the traditional Turing machine model.

The *second machine class* consists of those (parallel or sequential) devices which satisfy the Parallel Computation Thesis with respect to the traditional, sequential Turing machine model.

We propose a Unified Invariance Thesis in the following terms:

**Definition 5.5.1:**

**A machine model $<M,T_m,H_m>$ is "reasonable" if every terminating M-computation that works within (t(n),h(n)) resources on inputs of size n works within (t(n),r(n)) resources for $\log(r(n))=O(t(n)+\log(n))$ and if every $<M,T_m,H_m>$-computation taking (t,h) resources for some acceptable pair of resource bounds (t,h) can be simulated by a $<DTM,X_{dtm},S_{dtm}>$-computation**

**taking (poly(t+log(h)),O(h+log(t))) resources and every <DTM,$X_{dtm}$,$S_{dtm}$>-computation taking (t,h) resources for some acceptable pair of resource bounds (t,h) can be simulated by an <M,$T_m$,$H_m$>-computation taking (poly(t+log(h)),O(h+log(t))) resources.**

The following property holds for any "reasonable" model:

**Theorem 5.5.2:**

*Let <$M,T_m,H_m$> be any "reasonable" simultaneous complexity model. Define the resource $R_m$ as $R_m=T_m*(H_m+n)$, where n is the size of the input. Then the simultaneous complexity model <$M,R_m,H_m$> belongs to the first machine class. In particular, letting $R_{dtm}=X_{dtm}*(S_{dtm}+n)$, <$DTM,R_{dtm},S_{dtm}$> belongs to the first machine class.*

**Proof:**

Again, we first prove the case for the <DTM,$X_{dtm}$,$S_{dtm}$>, then use the "reasonability" of the model in question to prove the general case. Since the resource workspace is common to the (classical) Invariance Thesis and the <DTM,$X_{dtm}$,$S_{dtm}$>-equivalent of the Unified Invariance Thesis, the only question is the relationship between runtime and the product of mode transitions and the sum of workspace and input size. It has been shown in chapter 2 that the product of mode transitions and the sum of workspace and input size is an upper bound for runtime. On the other hand, both the workspace and the number of mode transitions are bounded by the runtime. As per the (classical) Invariance Thesis, only those computations that read the entire input (and hence take $\Omega(n)$ runtime) are relevant. Thus the square of the runtime is an upper bound (modulo constant multiplicative and additive factors) for the product of mode transitions and the sum of workspace and input size. This shows the polynomial equivalence of runtime and the new resource $R_m$ and the simultaneous linear equivalence of workspace with itself, thus proving that the simultaneous complexity model <DTM,$R_{dtm}$,$S_{dtm}$> belongs to the first machine class. It only remains to note that the "reasonability" of the simultaneous complexity model <M,$T_m$,$H_m$> together with the above result proves the theorem in the general case.

∎

We propose a conjecture in the following terms:

**Conjecture 5.5.3:**

*$<PRAM, T_{pram}, H_{pram}>$ is a "reasonable" simultaneous complexity model in the sense of the Unified Invariance Thesis.*

The following property holds for any "reasonable" model if the conjecture is true:

**Theorem 5.5.4:** *If the conjecture is true, then every "reasonable" simultaneous complexity model $<M, T_m, H_m>$ belongs to the second machine class. In particular, $<DTM, X_{dtm}, S_{dtm}>$ belongs to the second machine class.*

**Proof:**

It has to be shown only that parallel time is polynomially equivalent to $H_m$, without any restriction on the other resource, namely hardware or $T_m$. But for $<DTM, X_{dtm}, S_{dtm}>$ this holds by virtue of the classical proof. So it is enough to prove that parallel time is polynomially equivalent to $S_{dtm}$, then use the "reasonability" of the $<DTM, X_{dtm}, S_{dtm}>$ and the $<M, T_m, H_m>$ models in the sense of the Unified Invariance Thesis to show that $S_{dtm}$ is linearly equivalent to $H_m$. Note that the Parallel Computation Thesis requires the equivalence to hold only for resource bounds that are $\Omega(log(n))$.

∎

# Chapter 6

# Concluding Remarks

## 6.1 The Meaning of "Mode"

It is traditional to view a sequential computation as a history, and a parallel algorithm is obtained when this history is partitioned into a small number of large subsequences that are characterised by internal parallelizability and differentiated from each other by external sequentiality. Thus, the major task is the establishment of discontinuity.

But "establishing discontinuities (in history) is not an easy task ...We may wish to draw a dividing-line; but any limit we set may perhaps be no more than an arbitrary division made in a constantly mobile whole. We may wish to mark off a period,...(but) where, in that case, would the cause of its existence lie? Or that of its subsequent disappearance and fall? What rule could it be obeying by both its existence and its disappearance? If it contains a principle of coherence within itself, whence could come the foreign element capable of rebutting it?...(Discontinuity) begins with an erosion from outside, from that space which is, for thought, on the other side, but in which it has never ceased to think from the very beginning" [Foucault1970]

By choosing representations to characterise parallelizable periods of history a sequential computation is inverted to form a chain of organic structures; sequential time is transformed from a linear sequence of basic elements to a linear sequence of well-structured subsequences.

The "mode" of a tape represents an assumption of obliviousness that is not necessarily warranted, though it may hold for long periods of time before breaking down. It implies a separation of automaton state and behaviour from the irreducible richness of tape data and makes possible a concise and easy-to-compute predictive model that permits the simulator to eliminate sequential dependence within a sweep. Since the actual tape contents are ignored by the lookahead, an apparent discontinuity occurs when the delta function "behaves inconsistently". It is important to note the illusory nature of this inconsistency: the delta function does not even know that it has a mode; the discontinuity lies in the eye of the historian.

The set of possible modes represents the ability of the simulator to theorise about patterns in sequential history. As such, it is interesting to question whether, and to what extent, the richness of this theory limits the ability to extract parallelism. The results presented in this work indicate that performance optimal modulo polylog factors can be obtained with a constant-memory real-time data-oblivious prediction rule. Can one do better by considering more complex rules?

The intuitive notion of "predictive model" was mentioned above. The essence of prediction is avoiding the explosion of possible futures. This rough idea is used in theoretical computer science in at least three contexts including the present one: predictability could mean compressibility or determinism or parallelizability while unpredictability may mean randomness or alternation or inherent sequentiality. The relationship of parallelism to alternation is well-known. Is there a relationship between parallelism and compression?

## 6.2 The Role of Descriptive Complexity

The role of descriptive complexity as a "high level programming language" and as a method of simplifying the "global memory access pattern" has been discussed in chapter 1 and chapter 3. Two further points need to be mentioned. First, the representation of resource bounds as syntactic attributes resulted in the elimination of the proof of efficiency as an entity distinct and separate from the proof of correctness of the algorithm which simulates PRAMs

on the TM. This may be contrasted to the proof (in chapter 2) of the algorithm which simulates TMs on the PRAM. Secondly, the reduction to standard form may be viewed as a "Replication Theorem" [Blelloch'sBook1990]; the importance of such a theorem for practical parallel computing is well recognised. This suggests an important research area for descriptive complexity might be the development parallel programming languages that are designed to facilitate such a theorem.

## 6.3 Functions Computed by Communication Networks

That the problem of processing first order queries on relational databases is equivalent to the problem of communicating data between processors in a parallel computing system has been dramatically demonstrated in chapter 4 by using interconnection network techniques to implement relational transformations efficiently. While the concerns of computer engineers in designing such networks are of course far more elaborate, this work backs up the argument of [PippengerHandbook1990] that upper and lower bound results for computational problems might be obtained from corresponding bounds for communication networks even when the computational problems and the means used to solve them involve activities other than communication. Our choice of the term "basic redistributive functions" is intended to suggest that these may form the base functions in an hierarchy of functions similar to the primitive recursive functions.

## 6.4 The Correspondence between External and Parallel Algorithms

An amusing idea that is suggested by these results is that for decades researchers might have been doing research in parallel computing without being aware of the fact. The development of massive core memories is a recent phenomenon; the very word "core" refers to electric windings with cores of magnetic materials that used hysteresis to store a bit of data.

Databases were (and still are) stored on tape spools and access to the data was sequential. As a result, researchers concentrated on development of "file organisations" and "update algorithms" that simultaneously satisfied two constraints: the amount of tape used was to be kept low and the number of tape start, stop and rewind operations was to be kept low. The great practical importance of these "external algorithms" has ensured that they remain in the textbooks even though most algorithm designers dismiss them as obsolete.

Now it is apparent that not only are they not obsolete; they are promising candidates for good parallel algorithms! Our results give a precise formal sense in which the correspondence between external and parallel algorithms holds. It is interesting that heapsort is as difficult to parallelise as it is to externalise, while mergesort has both external and parallel versions. The problem of designing a nearly linear cost NC algorithm for transitive closure is open; path finding in graphs is notoriously a difficult problem in database applications. Database researchers have traditionally been critical of algorithms designed to be efficient on the RAM. There was once an expectation that "file theory" would evolve as a separate area of research; unfortunately the lack of uniformity in external storage architectures prevented this. It is thus possible that the appropriate complexity theory for databases is parallel theory rather than sequential theory; good file organizations correspond to good parallel data structures and good external update algorithms correspond to good parallel update algorithms in a uniform way. This would mean that database researchers are in their own way experts in parallel computing whether they know it or not!

## 6.5 Functional Dependencies and Recursive Query Processing

Could the paradigms of parallel algorithm design find application in database query processing? An example is evaluation of recursive queries. Since a directed graph with vertices restricted to outdegree 1 may be thought of as a functional dependency, and very efficient parallel algorithms for finding least common ancestors in such graphs are known [Tsin1986],

this seems possible at least in some speccial cases.

Recursive database queries can be defined by datalog programs or by fixpoint equations, both of which extend relational calculus. In the most general framework, such queries have high complexity. Therefore, it is important to recognize commonly occuring subclasses of recursions that can be evaluated efficiently using algorithms specially tailored to these subclasses. Besides the case in which complete materialisation of recursive relations is required, it is often the case that in the query some selection is applied to the recursive relation. In general, it is not possible to transpose the selection and recursion operations, and complete materialisation of the recursive relation becomes necessary. In [SippuSoisalon-Soininen1988], a class of generalised binary composition operators was proposed. Operators in this class are defined in terms of relational operations, are associative and have the property that the transitive closure of a first normal form (1NF) relation R with respect to such an operator can be computed in polynomial time by an algorithm which performs only O(polylog(|R|)) file operations (with a suitable definition of "file operation". For example a index or sort command on a file of size n counts as O(log(n)) file operations while a sequence of next-record commands or a sequence of previous-record commands or a rewind command counts as one file operation.) However, when a selection is applied to the closure, apparently either the entire relation has to be materialised, or the number of file operations cannot be restricted to O(polylog(|R|)).

A selection query on the generalised transitive closure is defined as follows [SippuSoisalon-Soininen1988]: Let the attribute columns of relations be ordered so that one can use "\$1" ("\$2","\$3" etc.) to refer to the $1^{st}$ (respectively $2^{nd}$,$3^{rd}$ etc.) component of a tuple while writing selections and projections. Assume selection predicates F are such that given an operand \$i of F, F may be written as $(F_1 \wedge F_2)$, where $F_1$ contains all relational predicates involving \$i and none of the others, and $F_1$ is not always true. Let $D=D_1 \times D_2 \times \cdots \times D_k$, $k>1$, be a finite product domain and let g be a binary operator on D; i.e. g maps pairs of relations over D into relations over D. g is a generalised composition if for all $R_1$ and $R_2$ over D, $g(R_1,R_2)=\pi_{\$i_1,\$i_2,\cdots,\$i_k}(\sigma_F(R_1 \times R_2))$ and the following conditions hold:

(a) For all $1 \leq j \leq k$, either $i_j = j$ or $i_j = k+j$.

(b) For all j, $1 \leq j \leq k$, if $\$i_j$ occurs in the selection predicate F, then F should be such that for the tuples selected by F, $\$j = \$(k+j)$ is true.

Let $R^+$ denote the least fixpoint of the equation $X = g(X,R) \cup R$. Let B be a subset of R. The selection query $R_B^+$ is defined as $R_B^+ = R^+ \cap \pi_{\$i_1,\$i_2,\cdots,\$i_k}(B \times R)$ and the selection query $R_B^-$ is defined as $R_B^- = R^+ \cap \pi_{\$i_1,\$i_2,\cdots,\$i_k}(R \times B)$

Since such selection queries occur frequently, it is worth exploring whether they can be evaluated more efficiently in special cases. Here we define a *generalised functional dependency* and give algorithms for the selection queries $R_B^+$ and $R_B^-$ which always work correctly, always perform only O(polylog(|R|)) file operations and are very efficient when the dependency holds. Specifically, when the dependency holds, they work in time $O(|R|\text{polylog}(|R|)+|B|*|R|)$. In the worst case they have the same time complexity as the algorithm for materialising the entire closure. Let a relation R over domain D be said to satisfy a generalised functional dependency with respect to a generalised composition operator g (denoted gfd(D,g,R)) if for every subset B of R and for every relation R′ over D, g(R′,B) can be computed in $O((|B|+|R'|)\text{polylog}(|B|+|R'|))$ time using O(polylog(|B|+|R′|)) file operations and the number of tuples in g(R′,B) is no more than the number of tuples in R′.

Example 1: Consider a personnel database containing a relation PERSONNEL with attributes EMP-ID, ALMA-MATER and MNGR-ID representing respectively the identification code of an EMPLOYEE, the name of the UNIVERSITY at which the employee studied last and the identification code of the employee's immediate supervisor. An employee has at most one immediate supervisor. All supervisors are employees. Given a relation OLDBOYS with attribute EMP-ID containing the identification codes of a set of employees, it is required to find all pairs (X,Y) such that X is the id of an employee, Y is the id of an employee who is directly or indirectly a manager of X, all employees in the chain of command from Y to X including Y and X are from the same university and Y occurs in relation OLDBOYS.

Let D be the domain of relation PERSONNEL and let g be the generalised composition operator

$g(R_1,R_2)=\pi_{R_1.EMP-ID,R_1.ALMA-MATER,R_2.MNGR-ID}$

$(\sigma_{R_2.EMP-ID=R_1.MNGR-ID \wedge R_2.ALMA-MATER=R_1.ALMA-MATER}(R_1 \times R_2))$.

It can be seen that gfd(D,g,PERSONNEL) holds. Let B be the result of the query

B=g(PERSONNEL,

$\pi_{PERSONNEL.EMP-ID,PERSONNEL.ALMA-MATER,PERSONNEL.MNGR-ID}$

$(\sigma_{PERSONNEL.EMP-ID=OLDBOYS.EMP-ID}(\text{OLDBOYS} \times \text{PERSONNEL})))$.

Let $\text{PERSONNEL}_B^-$ be a selection query as defined above. Then the required output can be obtained as $\pi_{EMP-ID,MNGR-ID}\text{PERSONNEL}_B^-$.

Example 2: Consider a personnel database containing a relation PERSONNEL with attributes EMP-ID, CADRE and MNGR-ID representing respectively the identification code of an EMPLOYEE, the cadre of the employee and the identification code of the employee's immediate supervisor. An employee has at most one immediate supervisor. All supervisors are employees. The cadre is either LABOUR or MANAGEMENT. Each employee who has a supervisor is directly or indirectly managed by an employee of the MANAGEMENT cadre. For all employees who have a supervisor, the juniormost employee of the MANAGEMENT cadre who manages the employee writes the employee's Annual Confidential Report. Given a relation TROUBLEMAKERS with attribute EMP-ID containing the identification codes of a set of employees, each having a supervisor, it is required to find all pairs (X,Y) such that X is the id of a troublemaker and Y is the id of the employee who writes the A.C.R. of X.

Let D be the domain of relation PERSONNEL and let g be the generalised composition operator

$g(R_1,R_2)=\pi_{R_1.EMP-ID,R_1.CADRE,R_2.MNGR-ID}$

$(\sigma_{R_2.EMP-ID=R_1.MNGR-ID \wedge R_2.CADRE \neq \text{"}MANAGEMENT\text{"}}(R_1 \times R_2))$

It can be seen that gfd(D,g,PERSONNEL) holds. Let B be the result of the query

$B=\pi_{PERSONNEL.EMP-ID,PERSONNEL.CADRE,PERSONNEL.MNGR-ID}$

$(\sigma_{PERSONNEL.EMP-ID=TROUBLEMAKERS.EMP-ID}$

(TROUBLEMAKERS×PERSONNEL)).

Let $R=\text{PERSONNEL}_B^+$ be a selection query as defined above. Then the required output

can be obtained as

$\pi_{R.EMP-ID,R.MNGR-ID}$

($\sigma_{PERSONNEL.EMP-ID=R.MNGR-ID \wedge PERSONNEL.CADRE="MANAGEMENT"}$

(R×PERSONNEL))

The following algorithm evaluates the selection query $R_B^+$ efficiently when the relation R satisfies a generalised functional dependency gfd(D,g,R): ($\Delta_1$ and $\Delta_2$ are intermediate relations)

begin

l:= precompute an upper bound (say $(k*|R|)^{2k}$ for 1NF relations R with k attribute columns) on the lengths of shortest derivations for those tuples for which a derivation exists;

$\Delta_1$:=$\Delta_2$:=R;

for $\lceil(log_2(l))\rceil$ iterations do

begin

$\Delta_1$:=g($\Delta_1$,$\Delta_1$);

$\Delta_2$:=$\Delta_2 \cup \Delta_1$;

end;

$R_B^+$:=$\Delta_1$:=B;

while($\Delta_1 \neq \phi$) do

begin

$\Delta_1$:=g($\Delta_1$,$\Delta_2$)-$\Delta_1$;

$R_B^+$:=$R_B^+ \cup \Delta_1$;

end;

end.

We call $\Delta_2$ a temporal index for R to distinguish it from ordinary indexes which we call spatial indexes.

The following algorithm evaluates the selection query $R_B^-$ efficiently when the relation R satisfies a generalised functional dependency gfd(D,g,R): ($\Delta_1$ and $\Delta_2$ are intermediate relations)

begin

l:= precompute an upper bound (say $(k*|R|)^{2k}$ for 1NF relations R with k attribute columns) on the lengths of shortest derivations for those tuples for which a derivation exists;

$\Delta_1$:=$\Delta_2$:=R;

for $\lceil(log_2(l))\rceil$ iterations do

begin

$\Delta_1$:=g($\Delta_1$,$\Delta_1$);

$\Delta_2$:=$\Delta_2 \cup \Delta_1$;

end;

$R_B^-$:=$\Delta_1$:=B;

while($\Delta_1 \neq \phi$) do

begin

$\Delta_1$:=g($\Delta_2$,$\Delta_1$)-$\Delta_1$;

$R_B^-$:=$R_B^- \cup \Delta_1$;

end;

end.

Note that these algorithms work correctly with cyclic data.

## 6.6 Query Language Extensions

Let $1NC_{dtm}$ be the class of problems solvable on DTMs using nearly linear workspace and polylog mode transitions. Let $1NCF_{dtm}$ be the corresponding function class. Similarly, let $1NC_{pram}$ be the class of problems solvable on PRAMs using nearly linear hardware and polylog parallel time. Let $1NCF_{pram}$ be the corresponding function class. It is clear that the classes defined on the DTM are subsumed by the corresponding classes defined on the PRAM. If the conjecture of chapter 5 is true, this inclusion is an identity and algorithms in this class are efficiently portable to all "reasonable" machine models. The classes $1NCF_{dtm}$ and $1NCF_{pram}$, along with LOGSPACEF, might prove useful while defining standards (com-

pleteness properties) for next generation query languages. While the need for extending the expressive power of query languages has been commented upon frequently, concern has focused on the fact that even quadratic complexity is unacceptable for database applications. These classes contain many problems of database interest not expressible in relational calculus (order statistics, categorical aggregation, path-finding in rooted trees) and calculi expressing these classes may prove to have the appropriate amount of expressive power for database applications. (Principles of independent interest like the Consistency Criterion would restrict the expressive power further.) This would also promote efficient portability to parallel environments.

Substantial research has been done already in the database field on new query languages that are more expressive than relationally complete languages. One of the most interesting of these, QBE, is very expressive, but is usually not implemented in full because it is too costly to do so. With hindsight, we can "explain" this by pointing out that QBE can express transitive closure of graphs, a query which is not known to be in either $1NCF_{pram}$ or LOGSPACEF. (The best known NC algorithm, based on Boolean Matrix Multiplication, has $O^*(m^{2.36})=O^*(n^{1.18})$ cost on graphs of m vertices, where the input, in adjacency matrix representation, is of size $n=m^2$. To qualify for inclusion in $1NCF_{pram}$, this must be reduced to $O^*(m^2)=O^*(n^1)$. See [CoppersmithWinograd1982])

The calculi developed in this study can in principle express queries which output relations (by leaving some variables unbound). However, by this relations are only indirectly represented, and it would be difficult to interface the retrieval language to the other parts of a practical query language (for example, creation, indexing and update of persistent data). Also, the strong assumption of order that is made here for the purpose of proving the complexity result is unacceptable for real query languages. It is therefore desirable to develop equivalent (and far more user-friendly) calculi that name and manipulate relations in the style of SQL, QUEL and QBE. It should be possible to assign intentionally defined relations to relation names within the scope of an iteration operator.

## 6.7 Suggestions for Future Work

On the descriptive complexity front, the following issues arise.

The power of FO+Y(LOG,LOG,BIN) seems difficult to characterize. Is the calculus capable of expressing PARITY and MAJORITY? It is clear that the class it expresses is subsumed by SC∩NC and strictly subsumed by Simultaneous-PolyTime-LinSpace.

The work of Abiteboul and Vianu [AbiteboulVianu1991b] on generic complexity shows that the complexity of a query can grow wildly when total ordering is removed. It would be interesting to know which queries remain expressible in the calculus FO+Y(T,S,V) when some or all of the machinery is removed (for example, one could remove some/all of the DOUBLE function, the SUC function, the BIN domain and the LOG domain) or replaced (say, by using the BIT predicate in place of the DOUBLE function).

Two invariance notions on space bounding functions, corresponding to (S=LIN,V=LOG) and (S∈{LIN,LOG},V=BIN) have been considered. Another invariance notion may be denoted by ignoring the space arity. It may be noted that other notions like being within a poly(log*(.)) factor of equality, being within a poly(alpha(.)) factor of equality (alpha(.) is the inverse Ackerman function) that seem to occur in recently published algorithms can be expressed by introducing additional piggyback domains as in chapter 5.

Characterization of calculi restricted in nesting-depth*space-arity product or in nesting-depth*time-arity product, more elegant operators equivalent to the Y-operator but with lesser primitive machinery and the extension of this explicit representation approach to other classes are other possibilities for study.

On the structural complexity front, our classes are uniform. Can similar non-uniform (P-uniform) class be defined and characterised? What relationships can be adduced between $NC^1$ (or LOGSPACE, which subsumes it) and $1NC_{dtm}$ (or $1NC_{pram}$, which subsumes it)?

On the classical complexity front, the following issue arises:

The results presented in this work, which arose from an attempt to unify the Invariance Thesis, the Parallel Computation Thesis, Pippenger's simulation result and the work on

"efficient" simulations between parallel models, are part of a (possibly hopeless) programme to recover the uniform, unfragmented perspective which computational complexity theory enjoyed in the 1960s because of which it was recognised both as a branch of mathematics and as a branch of engineering. An obvious next step is to incorporate the distinction between computing elements (feedforward) and persistent memory (feedback) and study serial case parallelism as triple resource bounded complexity. Does there exist a variant of Turing machines capable of handling the serial case? See also the Pipelined Computation Thesis of [Wiedermann1992].

On the recursive query processing front, the notion of using (variants of) functional dependency constraints for efficient query evaluation can be extended to different subclasses of datalog programs. Algorithms could be designed for use only when the constraint is known to hold or they could be designed to be always correct and to work more efficiently when the constraint happens to hold.

On the query language extensions front, the complexity of database query language features needs to be studied. Since a total ordering is in general not available in end-user query languages, a collection of special-purpose operators like the unique and group by operators of SQL, aggregation functions, etc. are used instead. Studying the expressiveness/complexity trade-off of these operators, in isolation and in combination with iteration operators might lead to the development of more expressive yet practical query languages.

While the need for fast business computing platforms is growing rapidly, parallel computing platforms are suffering from a lack of non-scientific software applications. As database files often are organised and COBOL programs often are written quite deliberately to use nearly linear amounts of tape and to minimize the number of stops and starts by the tape drive, further work might lead to a way of porting the large established base of business computing software to parallel computing environments (by detecting parallelism in COBOL programs in the spirit of these results) with nearly the same cost and nearly optimal speedup.

## References


[AbiteboulVianu1990] S.Abiteboul & V.Vianu, Procedural languages for database queries and updates, JCSS 41, 1990, pp.181-229.

[AbiteboulVianu1991a] S.Abiteboul & V.Vianu, Datalog extensions for database queries and updates, JCSS 43, 1991, pp.62-124.

[AbiteboulVianu1991b] S.Abiteboul & V.Vianu, Generic computation and its complexity, Proc. ACM STOC, 1991, pp.209-219.

[AhoUllman1979] Aho and Ullman, Universality of Data Retrieval Languages, 6th POPL, 1977.

[Allender1985] E.W.Allender, Invertible functions, Ph.D. thesis, Georgia Inst. Tech., Sept. 1985.

[Allender1989] E.W.Allender, P-uniform circuit complexity, JACM 36, 1989, pp.912-928.

[AltHagerupMehlhornPreparata1987] H.Alt, T.Hagerup, K.Mehlhorn & F.P.Preparata, Deterministic simulation of idealised parallel computers on more realistic ones, SIAM J. Comput., Vol. 16, No. 5, Oct. 1987, pp.808-835.

[BarringtonImmermanStraubing1988] D.A.M.Barrington, N.Immerman & H.Straubing, On uniformity within $NC^1$, Proc. $3^{rd}$ Struc. in Complexity Th., 1988, later appeared in JCSS 41, 1990, pp.274-306.

[Batcher1968] K.E.Batcher, Sorting networks and their applications, in Proc. SJCC, AFIPS Press, Montvale, NJ, Vol. 32, 1968, pp.307-314.

[Blelloch'sBook1990] G.E.Blelloch, Vector models for data-parallel computing, MIT Press, Cambridge, 1990.

[Blelloch1989] Scans as primitive parallel operations, IEEE Trans. on Comput. 38(11), 1989, pp.1526-1538.

[Borodin1977] A.Borodin, On relating Time and Space to Size and

Depth, SIAM J. Comput., Vol. 6, No. 4, Dec. 1977, pp.733-744.

[BorodinHopcroft1985] A.Borodin & J.E.Hopcroft, Routing, merging and sorting on parallel models of computation, JCSS 30, 1985, pp.130-145.

[CaiFurerImmerman1989] J.Cai, M.Furer & N.Immerman, An optimal lower bound on the number of variables for graph identification, Proc. $30^{th}$ IEEE FOCS, 1989, pp.612-617.

[Chandra1988] A.K.Chandra, Theory of database queries, Proc. ACM Symp. PODS, 1988, pp.1-9.

[ChandraHarel1980] A.K.Chandra & D.Harel, Computable queries for relational data bases, JCSS 21, 1980, pp.156-178.

[ChandraHarel1982] A.K.Chandra & D.Harel, Structure and Complexity of Relational Queries, JCSS 25, 1982, pp.99-128; (prelim. ver. 21st FOCS 1980).

[ComptonLaFlamme1988] K.J.Compton & C.Laflamme, An algebra and a logic for $NC^1$, Proc. IEEE LICS, 1988, pp.12-21.

[Cook1985] S.A.Cook, A taxonomy of problems with fast parallel algorithms, Inf. & Ctrl. 64, 1985, pp.2-22.

[CookHoover1985] S.A.Cook & H.J.Hoover, A depth-universal circuit, SIAM J. Comput., Vol 14, No. 4, Nov. 1985, pp.833-839.

[CoppersmithWinograd1982] D.Coppersmith & S.Winograd, On the asymptotic complexity of matrix multiplication, SIAM J. Comput., Vol. 11, No. 3, Aug. 1982, pp.472-492.

[Davio1981] M.Davio, Kronecker products and shuffle algebra, IEEE Trans. on Comput., Vol. C-30, No. 2, Feb. 1981, pp.116-125.

[DublishMaheshwari1989] P.Dublish & S.N.Maheshwari, Expressibility of Bounded-

Arity Fixed-Point Query Hierarchies, Proc. 9th ACM Symp. PODS, 1989.

[Dymond1980] P.W.Dymond, Simultaneous resource bounds and parallel computation, Ph.D. thesis, Univ. Toronto, Aug. 1980.

[Fagin1974] R.Fagin, Generalised first-order spectra and polynomial time recognizable sets, in Complexity of Computation (ed. R.Karp), SIAM-AMS Proc. 7, 1974, pp.27-41.

[Feng1974] T.-Y.Feng, Data manipulating functions in parallel processors and their implementations, IEEE Trans. on Comput., Vol. C-23, No. 3, Mar. 1974, pp.309-318.

[FichRagdeWigderson1988] F.E.Fich, P.Ragde & A.Wigderson, Relations between concurrent-write models of parallel computation, SIAM J. Comput., Vol. 17, No. 3, Jun. 1988, pp.606-627.

[Foucault1970] M.Foucault, The order of things: An archaeology of the human sciences, Tavistock Publications Ltd., London 1970. Translated from the French original, Le mots et les choses, Editions Gallimard, 1966.

[Fraser1976] D.Fraser, Array permutation by index-digit permutation, JACM, Vol. 23, No. 2, Apr. 1976, pp.298-309.

[GalilPaul1983] Z.Galil & W.J.Paul, An efficient general-purpose parallel computer, JACM, Vol. 30, No. 2, Apr. 1983, pp.360-387.

[Goldschlager1982] L.M.Goldschlager, A universal interconnection pattern for parallel computers, JACM, Vol. 29, No. 3, Jul. 1982, pp.1073-1086.

[GurevichShelah1985] Y.Gurevich and S.Shelah, Fixed Point Extensions of First-Order Logic, Proc. IEEE FOCS, 1985.

[Hartmanis1968] J.Hartmanis, Tape-reversal bounded Turing machine computations, JCSS 2, 1968, pp.117-135.

[HennieStearns1966] F.C.Hennie & R.E.Stearns, Two-way simulation of multitape Turing machines, JACM 13, 1966, pp.533-546.

[HerleyBilardi1988] K.T.Herley & G.Bilardi, Deterministic simulations of PRAMs on bounded degree networks, Tech. Rep. 88-9511, Dept. of Comp. Sc., Cornell Univ., Nov. 1988.

[Hong1984a] J.W.Hong, On similarity and duality of computation, Inf. & Ctrl., Vol. 62, Nos. 2/3, Aug./Sept. 1984, pp.109-128.

[Hong1984b] J.W.Hong, A tradeoff theorem for space and reversal, TCS 32, 1984, pp.221-224.

[Immerman1982] N.Immerman, Relational Queries Computable in Polynomial Time, STOC 1982 (later appeared in Inf. & Ctrl. 68, Mar. 1986, pp.86-104).

[Immerman1986] N.Immerman, Languages which capture complexity classes, Yale Univ. Tech. Rep., Jan. 1986, later appeared in SIAM J. Comput., Vol. 16, No. 4, Aug. 1987, pp.760-778.

[Immerman1987a] N.Immerman, Expressibility as a complexity measure: results and directions, Tech. Rep., later appeared in 2nd Struc. in Complexity Th., 1987, pp.194-202.

[Immerman1987b] N.Immerman, 1987, Expressibility and Parallel Complexity, Tech. Rep. YALEU/DCS/TR-546, later appeared in SIAM J. Computing, Vol. 18, No. 3, 1989, pp.625-638.

[Immerman1991] N.Immerman, DSPACE[$n^k$]=VAR[k+1], IEEE Structures, 1991, pp.334-340.

[ImmermanLander1990] N.Immerman & E.Lander, Describing graphs: a first order approach to graph canonization, in A.Selman (ed.), Complexity theory retrospective (in honor of Juris Hartmanis), Springer, New York, 1990, pp.59-81.

[KamedaVollmar1970] T.Kameda & R.Vollmar, Note on tape reversal complexity of languages, Inf. & Ctrl. 17, 1970, pp.203-215.

[KarpRamachandranHandbook1990] R.M.Karp & V.Ramachandran, Chapter 17: Parallel algorithms for shared-memory machines, pp.869-941, in J.v.Leeuwen (ed.), Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity, Elsevier, Amsterdam, 1990.

[Lawrie1975] D.H.Lawrie, Access and alignment of data in an array processor, IEEE Trans. on Comput., Vol. C-24, No. 12, Dec. 1975, pp.1145-1155.

[Lenfant1978] J.Lenfant, Parallel permutations of data: a Benes network control algorithm for frequently used permutations, IEEE Trans. on Comput., Vol. C-27, No. 7, Jul. 1978, pp.637-647.

[LevPippengerValiant1981] G.F.Lev, N.Pippenger & L.G.Valiant, A fast parallel algorithm for routing in permutation networks, IEEE Trans. on Comput., Vol. C-30, No. 2, Feb. 1981, pp.93-100.

[Lindell1987] S.Lindell, The logical complexity of queries on unordered graphs, Ph.D. thesis, Dept. of Comp. Sc., UCLA, 1987.

[NassimiSahni1981a] D.Nassimi & S.Sahni, Data broadcasting in SIMD computers, IEEE Trans. on Comput., Vol. C-30, No. 2, Feb. 1981, pp.101-107.

[NassimiSahni1981b] D.Nassimi & S.Sahni, A self-routing Benes network and parallel permutation algorithms, IEEE Trans. on Comput., Vol. C-30, No. 5, May 1981, pp.332-340.

[NassimiSahni1982] D.Nassimi & S.Sahni, Optimal BPC permutations on a cube connected SIMD computer, IEEE Trans. on Comput., Vol. C-31, No. 4, Apr. 1982, pp.338-341.

[Parberry1986] I.Parberry, Parallel speedup of sequential machines: a defense of the parallel computation thesis, SIGACT News, Vol. 18, No. 1, 1986, pp.54-67.

[ParberrySchnitger1988] I.Parberry & G.Schnitger, Parallel computation with threshold functions, JCSS 36, 1988, pp.278-302.

[Parker1980] D.S.Parker Jr., Notes on shuffle/exchange-type switching networks, IEEE Trans. on Comput., Vol. C-29, No. 3, Mar. 1980, pp.213-222.

[Pippenger1979] N.Pippenger, On simultaneous resource bounds (preliminary version), Proc. FOCS, 1979, pp.307-311.

[Pippenger1987] N.Pippenger, Sorting and selecting in rounds, SIAM J. Comput., Vol. 16, No. 6, Dec. 1987, pp.1032-1038.

[PippengerHandbook1990] N.Pippenger, Chapter 15: Communication networks, pp.805-833, in J.v.Leeuwen (ed.), Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity, Elsevier, Amsterdam, 1990.

[PrattStockmeyer1976] V.R.Pratt & L.J.Stockmeyer, A characterisation of the power of vector machines, JCSS 12, 1976, pp.198-221.

[Ruzzo1981] W.L.Ruzzo, On uniform circuit complexity, JCSS 22, 1981, pp.365-383.

[Siegel'sBook1990] H.J.Siegel, Interconnection networks for large-scale parallel processing: theory and case studies, $2^{nd}$ ed., McGraw-Hill, New York, 1990.

[Siegel1977] H.J.Siegel, Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks, IEEE Trans. on Comput., Vol. C-26, No. 2, Feb. 1977, pp.153-161.

[Siegel1979] H.J.Siegel, A model of SIMD machines and a compari-

son of various interconnection networks, IEEE Trans. on Comput., Vol. C-28, No. 12, Dec. 1979, pp.907-917.

[SippuSoisalon-Soininen1988] S.Sippu & E.Soisalon-Soininen, A generalized transitive closure for relational queries, Proc. ACM Symp. PODS, 1988, pp.325-332.

[Stewart1989] I.A.Stewart, Using the Hamiltonian path operator to capture NP, Proc. $2^{nd}$ Int. Conf. Computing & Information, May 1989, LNCS, Springer-Verlag.

[StockmeyerVishkin1984] L.J.Stockmeyer and U.Vishkin, Simulation of Parallel Random Access Machines by Circuits, SIAM J. Comput., Vol. 13, No. 2, 1984, pp.409-422.

[Stone1971] H.S.Stone, Parallel processing with the perfect shuffle, IEEE Trans. on Comput., Vol. C-20, No. 2, Feb. 1971, pp.153-161.

[Thompson1978] C.D.Thompson, Generalized connection networks for parallel processor intercommunication, IEEE Trans. on Comput., Vol. C-27, No. 12, Dec. 1978, pp.1119-1125.

[Tsin1986] Y.H.Tsin, Finding lowest common ancestors in parallel, IEEE Trans. on Comput. 35, 1986, pp.764-769.

[Valiant1976] L.G.Valiant, Universal circuits, Proc. $8^{th}$ ACM STOC, 1976, pp.196-203.

[ValiantHandbook1990] L.G.Valiant, Chapter 18: General purpose parallel architectures, pp.943-971, in J.v.Leeuwen (ed.), Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity, Elsevier, Amsterdam, 1990.

[VanEmdeBoasHandbook1990] P.v.E.Boas, Chapter 1: Machine models and simulations, pp.1-66, in J.v.Leeuwen (ed.), Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity,

Elsevier, Amsterdam, 1990.

[Vardi1982] M.Vardi, Complexity of relational query languages, $14^{th}$ STOC, 1982, pp.137-146.

[Wiedermann1992] J.Wiedermann, Weak parallèl machines: a new class of physically feasible parallel machine models, Proc. MFCS, 1992, Springer LNCS 629, pp.95-111.

[WuRosenfeld1981] A.Y.Wu & A.Rosenfeld, SIMD machines and cellular d-graph automata, IEEE Trans. on Comput., Vol. C-30, No. 5, May 1981, pp.370-372.

[YewLawrie1981] P.-C.Yew & D.H.Lawrie, An easily controlled network for frequently used permutations, IEEE Trans. on Comput., Vol. C-30, No. 4, Apr. 1981, pp.296-298.

[Zloof1977] M.M.Zloof, Query-By-Example: a database language, IBM Sys. J., Vol. 16, No. 4, 1977, pp.324-343.